Mixed-Initiative Visual Programming Paradigms for Authoring Sensor-Based Systems

Raf Ramakers
Hasselt University - tUL - imec
Expertise Centre for Digital Media
Diepenbeek, Belgium
raf.ramakers@uhasselt.be

Kris Luyten
Hasselt University - tUL - imec
Expertise Centre for Digital Media
Diepenbeek, Belgium
kris.luyten@uhasselt.be

Abstract

Most users of computing devices are non-programmers and are limited to passive consumers of the technology that is made available to them. While research on end-user design environments resulted in procedures to automatically generate design and electronic aspects of sensor-based systems, automated code generation is limited to simple repetitive tasks. In this position paper, we explore mixed-initiative approaches to assist users in specifying logic for sensor-based systems. By combining visual programming and machine learning concepts, our approach presents a tight interactive loop in which the user and system both take turns in producing, evaluating, and modifying logic behavior.

Author Keywords

Visual Programming; Design Environments; Inductive Programming; Programming-By-Demonstration; Ubiquitous Computing.

ACM Classification Keywords

H.5.2 [[Information interfaces and presentation]]: :User Interfaces

Introduction

Advancements in thin-film electronics and digital fabrication technologies make it possible to seamlessly embed interactivity in physical artifacts. Recently, there is an increasing interest in end-user design environments that enable people without a technical background to author such ubiquitous interfaces [8, 9, 11]. These kind of design environment offer an end-to-end approach by guiding the user and streamlining the process to make ubiquitous computing interfaces. Although it is feasible to abstract and automate many graphical modeling and electronic design tasks [8, 9, 11], accurately generating the underlying logic that represents the desired behavior is still extremely challenging. Especially so for interactive systems, since the interpretation of the computer should match with the expectations of the users. Therefore, non-programmers are required to learn programming skills to successfully translate the envisioned workings of the system into a sequence of statements.

To facilitate expressing the behavior of interactive system by non-programmers, researchers investigated visual programming and programming-by-demonstration methodologies. Visual programming approaches [7] facilitate the process of specifying logic behavior by offering visual building blocks. While these approaches mitigate syntax issues, a basic understanding of different programming constructs still needs to be learned. Especially when specifying temporal behavior, often present in sensor-based systems, constructs, such as variables, loops, functions, and timers are required. Examples include, actions triggered after a temperature change over time, a button that has to be pressed fast or long enough, an LED or audio signal triggered at a certain frequency, conditions that need to be completed in a certain order or at the same time. Specifying such behaviors has a steep learning curve and is often cumbersome for non-experts. In contrast, programming-by-demonstration [5] approaches automate the entire programming procedure by synthesizing logic constructs from input and output examples demonstrated by the user. While convenient for first-time programmers, specifying precise relations and detailed timing properties is cumbersome and requires demonstrating many positive and negative examples.

In this position paper, we present a mixed-initiative approach to assist users in specifying logic for sensor-based systems. By combining visual programming and machine learning concepts used in programming-by-demonstration, our approach presents a tight interactive loop in which the user and system both take turns in producing, evaluating, and modifying logic behavior. Our work builds on top of a novel domain-specific visual programming paradigm, Pulsation, that is tailored for specifying the behavior of sensor-based systems. If appropriate, we can demonstration of our Pulsation logic specification technique during the workshop.

Background and Related Work

This work draws from, and builds upon work in visual programming and inductive programming techniques.

To allow users without a programming background to specify logic behavior, researchers investigated visual programming techniques [7]. Unlike visual programming approaches to avoid syntax errors, such as Scratch [10], Pulsation facilitates authoring the behavior of sensor-based systems as every logic construct offers precise control over timing properties. In contrast, Scratch [10] or visual programming approaches supported by LEGO Mindstorms [1] require users to construct complex algorithms using timers in order to match or produce patterns over time. Amongst the visual programming approaches that target specific domains, such as PureData[1] for audio processing or Nodebox[2] graphical renderings, LabView [12] and programming approaches

[1] https://puredata.info
[2] https://www.nodebox.net

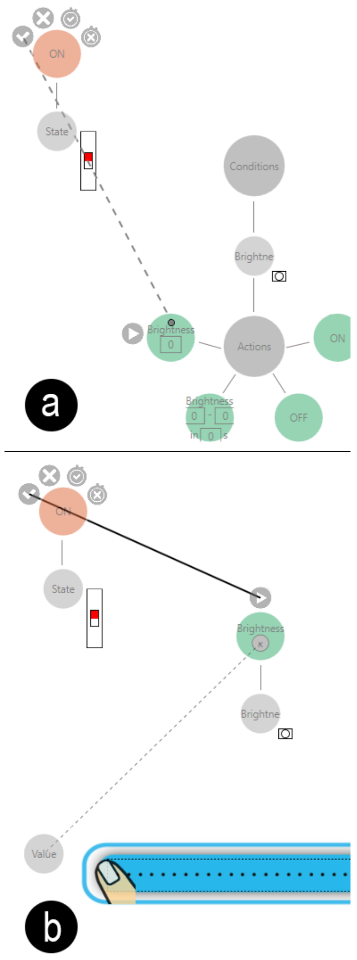

Figure 1: Continuously assigning the value of a slider to the brightness of an LED when a switch is in the on-state.

in Loxone[3] are optimized for sensor-based systems, similar to Pulsation. However, both systems target experts and thus require users to follow extensive tutorials. The IFTTT system[4] on the other hand, simplifies programming to elementary conditional behavior but has limited expressive power.

To automate the programming process, researchers looked into techniques to synthesize code from users' provided example. This methodology, often referred to as inductive programming [4], induces logic behavior consistent with the user's specified examples of the intended behavior. For example the Flash Fill feature [3] in Microsoft Excel 2013 mitigates repetitive string transformations by automatically generating string processing scripts from one or more user-provided examples. In ubiquitous computing, one can also demonstrate examples directly with sensors in the real-world. These kind of programming approaches are often referred as programming-by-demonstration or programming-by-example [5, 2].

Pulsation

In contrast to visual programming languages, such as Scratch [10] or LEGO Mindstorms [1] that rely on abstract building blocks, Pulsation allows users to specify behavior directly on top of components using visual links (Figure 1). Although visual links facilitate the visibility and overview of logic constructs for non-experts, it is unclear how simple links can translate to coherent specifications in a grammar. Even for simple components, such as linking a pushbutton to an LED, many different relationships are possible including, turning the LED on when pushed and turning it off when pushed again (toggle), turning the LED immediately off when the button is released. Alternatively, the LED can fade in over time or blink. Therefore Pulsation supports an interaction paradigm for constructing a detailed logic graph to precisely link logic constructs (Figure 1). In this section, we first discuss the different constructs available in Pulsation (grammar). Then we show how the visual programming paradigm facilitates instantiating grammar constructs. Pulsation grammar can be tested by users in a run-time environment.

[3]http://www.loxone.com
[4]https://ifttt.com

Grammar

The majority of behavior in sensor-based systems is influenced by some sort of temporal behavior. The Pulsation logic grammar is therefore tailored for specifying behavior that requires precise control over time. The grammar consists of five constructs: variables, conditions, actions, and events. Conditions are boolean expressions that evaluate the state of variables, actions change variables, and events of conditions trigger events of actions. To allow for loops and encapsulations, Pulsation supports aggregate and derived constructs that are specific to conditions and actions. Condition aggregates consist of multiple conditions (conjunction/disjunction), and a temporal relationship (e.g. completed simultaneously, or ordered, etc.). Derived conditions calculate a specific derived parameter from the child condition, such as the number of times the child condition is completed (loop/repeater), duration of time to complete the condition, percentage of child conditions completed (progress), etc. Similarly, action aggregates consist of multiple actions and allow for specifying delays. Derived actions are regulated by parameters e.g. repeating actions a number of times or executing only a limited number of actions of an aggregate action.

Visual Logic Specifications

Pulsation offers an interaction paradigms for constructing a detailed logic graph to precisely link conditions and actions. Figure 1 demonstrates the basics: (a) Clicking on an elec-

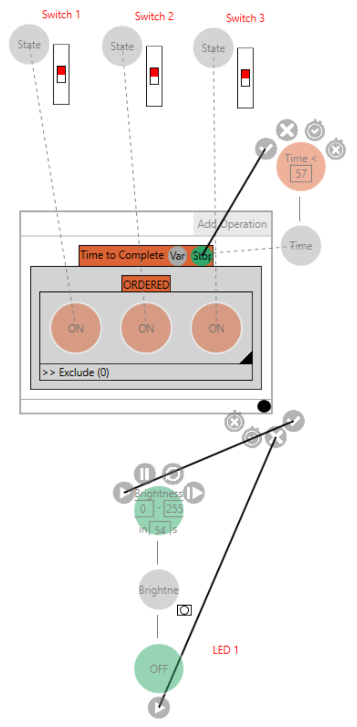

Figure 2: Turning all switches to on within 5 seconds will start fading the LED.

tronic component reveals a hierarchical radial menu consisting of conditions and actions that relate to the variable associated with that component (e.g. brightness variable of an LED). This includes basic conditions and actions (e.g. brightness equals x, turn LED on), as well as derived conditions and actions (e.g. LED on for x seconds, fade LED from x to y in t seconds). While hovering over these conditions and actions, associated events appear. Events of conditions are linked to events of actions by drawing visual links using a drag-and-drop interaction style. (b) In this example, we specify that the value of a slider is continuously assigned to the brightness of an LED as long as the switch is in the on-state.

Radial menus presented in the previous paragraph offer conditions and actions in the context of a single variable. Advanced conditions and actions consisting of multiple variables, are authored using a condition and action composer. Using these advanced composers, aggregate conditions/actions are nested to support precise control of timing parameters and relations between variables. Figure 2 shows a condition composer consisting of a nested aggregate and derived condition to specify that three switches have to be turned on, in a sequential order, within 5 seconds to make the LED fade in.

Mixed-Initiative Programming Approaches

Although Pulsation facilitates the process of specifying the behavior of sensor-based systems, users still need to correctly translate the desired behavior to logic rules. Novices oftentimes lack computational thinking skills and end up with incorrect or incomplete logic specifications. For example, it is challenging for first-time users to identify conflicting behavior or infer when variables are needed to store intermediate results, when to stop or pause the execution of logic, and when to reset a variable. In earlier versions of Pulsation [9], we identified common situations in which it is desired to automatically introduce additional actions, such as reverting actions when a condition is not satisfied anymore. However during tests, we oftentimes encountered situations in which these automatic actions limited the expressive power, for example, in cases were reversing the action was not appropriate or the inverse operation was ambiguous to determine.

To assist users, without limiting the expressive power of Pulsation, we introduce mixed-initiative programming techniques to suggest logic addition, corrections, or alternatives that are appropriate in the current context. To infer additional logic constructs from user specified logic behavior, we consider probabilistic modeling, machine learning, and inductive programming strategies. Using these techniques, the system suggests logic constructs to the user and updates its recommendations every time the user adds Pulsation logic. The suggested additions can be updated or removed after testing in the Pulsation run-time environment. As such, there is a continuous interplay between the user and the system in order to facilitate authoring behavior of sensor-based systems.

Example Scenarios and Techniques

We identified three situations in which mixed-initiative approaches could significantly help users in composing logic behavior for sensor-based systems:

Suggesting Logic Additions

Given the behavior already specified, some logic constructs are more likely to be required than others in order to complete the behavior of a system. By analyzing the Pulsation logic already specified, the system can automatically recommend logic additions that users might forget otherwise. These kind of recommendations can be derived from previously specified Pulsation programs (from the same or different users) using stochastic models (e.g. Markov widget in Grasshopper Rhino[5]) or classification algorithms.

For example, in conditional behavior it is often desired that actions are reversed once the conditions are not satisfied anymore. First-time programmers oftentimes forget such inverse operations. Although inverse actions are convenient to add for very simple condition-action triggers (e.g. turning LED on when button is pressed), inverse actions are ambiguous when advanced actions or many components are involved. For example, the inverse of fading in the brightness of an LED over time, could be a fade-out operation or a reset to its original brightness level. Such a reverse operation is not desired at all when the system is in a certain state when the condition (e.g. logging in on a numeric pad) is completed. Similarly, novices often forget to reset stop/pause execution of actions, or reset variables and states. By leveraging the knowledge of previously specified programs, the system can suggest one or multiple logic constructs that users can build upon further.

Resolving Incomplete Specifications
Informal tests with Pulsation revealed that it is oftentimes challenging for first-time programmers to compose detailed actions and conditions, especially when variable parameters or sub-actions (e.g. pause/stop) are involved. With probabilistic models in place, one could directly interlink components after which the Pulsation synthesizer initiates the most appropriate conditions and actions. For example, when linking a rotary dial or linear slider to a light or LED, the system can propose logic for realizing a dimmer. When connecting a switch to this configuration, the system could automatically recommend logic for realizing a master switch in the system. Employing these kind of intelligent decisions, ensures that the system is always meaningful and operational while the user can refine logic behavior iteratively after testing.

[5]http://www.grasshopper3d.com

Identifying and Resolving Incorrect Specifications
When systems do not work as expected, it is very challenging for non-programmers to inspect the system and identify the incorrect or missing logic construct. We propose an inductive programming strategy [4] were the system infers new or alters existing logic constructs to resolve the behavior. In our approach, the user annotates and corrects the state of components in the run-time environment when an abnormality occurs. For example, the user annotates that an alarm is off whereas it should be turned on. The system will synthesize new logic constructs consistent with the execution of the system and the corrections provided by the user. The inference may draw solely on the input/output data and corrections in the current system or use additional auxiliary concepts from background knowledge synthesized from other user-defined behavior. When multiple solutions are viable, logic rules can be ranked based on their simplicity, size, or generalizability [3].

Challenges
Although probabilistic modeling, machine learning, and inductive programming strategies are very promising to realize mixed-initiative visual programming approaches, implementing these concepts comes with a few challenges:

- Logic constructs specified by the user require careful consideration before being synthesized as training data. Automatically verifying the meaning and stability of logic behavior is however challenging.
- One very important property of authoring environments is offering a high ceiling to allow for many design variations.

Logic synthesizers thus need to operate in a slightly different context every time. Example logic behavior that is qualified to be used as training data therefore needs to be abstracted and converted to auxiliary concepts first. Defining which auxiliary concepts are substantial enough to be used as background knowledge is non-trivial. Especially for inductive programming strategies, too many auxiliary concepts make it hard to select the appropriate concepts in the induction step whereas to little auxiliary concepts requires new concepts to be invented by the synthesizer [6]

- It is challenging for the system to determine when to offer logic suggestions. When recommending logic constructs early on in the authoring process, the recommender system might not have enough context and knowledge to suggest appropriate behavior. Deferring the suggestions too long might deteriorate the user's ability to author the behavior of a system.

Conclusion

As ubiquitous computing systems and IoT technologies continues to rise in popularity, users will desire more control over the behavior of these devices. In this position paper, we introduced mixed-initiative programming paradigms that facilitate and guide the user in specifying logic for sensor-based systems. We discussed how our domain specific visual programming approach, Pulsation, can be enriched with probabilistic models, machine learning techniques, and inductive programming strategies to bridge the gap for non-programmers to specify logic behavior. We discussed the novel opportunities of these approaches as well as the challenges for future research. If appropriate, we can demonstration of our Pulsation logic specification technique during the workshop.

Acknowledgements

Funding for this research was provided by the Fund For Scientific Research Flanders (F.W.O. Vlaanderen, project number G0E7317N).

References

[1] David J Barnes. 2002. Teaching introductory Java through LEGO MINDSTORMS models. In *Proc. SIGCSE Bulletin*, Vol. 34. ACM, 147–151.

[2] Adam Fourney and Michael Terry. 2012. PICL: Portable In-circuit Learner. In *Proc. UIST '12*. ACM, 569–578.

[3] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 317–330.

[4] Sumit Gulwani, Jose Hernandez-Orallo, Emanuel Kitzelmann, , and others. 2015. Inductive programming meets the real world. *Commun. ACM* 58, 11 (2015), 90–99.

[5] Björn Hartmann, Leith Abdulla, Manas Mittal, and Scott R. Klemmer. 2007. Authoring Sensor-based Interactions by Demonstration with Direct Manipulation and Pattern Recognition. In *Proc. CHI '07*. ACM, 145–154.

[6] José Hernández-Orallo. 2013. *Deep knowledge: Inductive programming as an answer*. Technical Report. Dagstuhl TR 13502.

[7] Caitlin Kelleher and Randy Pausch. 2005. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Comput. Surv.* 37, 2 (June 2005), 83–137.

[8] Raf Ramakers, Fraser Anderson, Tovi Grossman, and George Fitzmaurice. 2016. RetroFab: A Design Tool for Retrofitting Physical Interfaces Using Actuators, Sensors and 3D Printing. In *Proc. CHI '16*. ACM, 409–419.

[9] Raf Ramakers, Kashyap Todi, and Kris Luyten. 2015. PaperPulse: An Integrated Approach for Embedding Electronics in Paper Designs. In *Proc. CHI '15*. ACM, 2457–2466.

[10] Mitchel Resnick, John Maloney, and others. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.

[11] Valkyrie Savage, Sean Follmer, Jingyi Li, and Björn Hartmann. 2015. Makers' Marks: Physical Markup for Designing and Fabricating Functional Objects. In *Proc. UIST '15*. ACM, 103–108.

[12] Lisa K Wells and Jeffrey Travis. 1996. *LabVIEW for everyone: graphical programming made even easier*. Prentice-Hall, Inc.