Complex Event Pattern Recognition Software for Multimodal User Interfaces

Aaron B. Phillips
Computer Science Department
Wheaton College
Aaron.B.Phillips@wheaton.edu

Jordan Kairys
Computer Science
Kalamazoo College
K00jk01@kzoo.edu

Will Fitzgerald
Institute for Information Technology – e-Business
National Research Council Canada
will.fitzgerald@pobox.com

Abstract

We describe how CERA, the Complex Event Recognition Architecture, was used to create a multimodal user interface for Skibbles, a memory game moderated by a mobile robot. We also announce the availability of open-source software that implements CERA, and how it can be used to build intelligent multimodal interfaces.

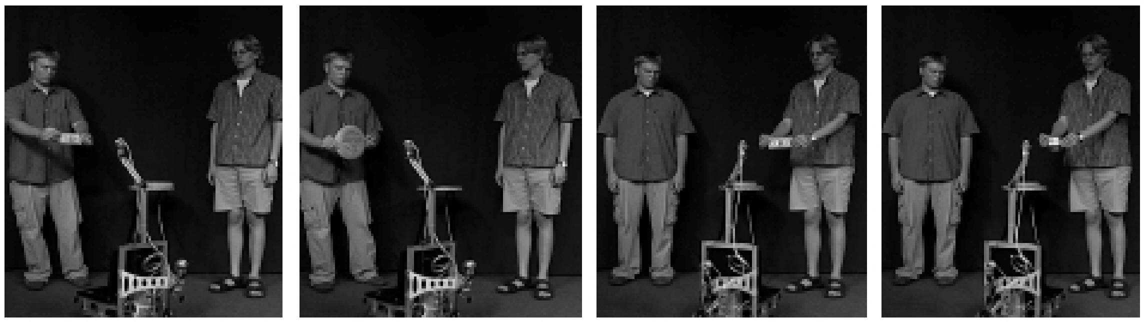
Introduction

As humans we experience the world through our five senses. Similarly, in computer science a multimodal interface is one that allows for input from many channels, also known as modes. The classic example of a multimodal interface is an in-car navigation system where the user touches the map and says, "Go here," simultaneously. Multimodal user interfaces, unlike traditional methods, allow for more natural human-computer interactions.

Typical multimodal user interfaces process each mode of interaction as a separate input stream. Each stream is individually analyzed to build a structured representation of its meaning. These representations are then combined together by some sort of integration module. However, CERA, the Complex Event Recognition Architecture, uses patterns to process input from any combination of modes together. System processing can use separate patterns within each mode when that makes sense, or patterns can combine events from different modes. In either case, the meaning of various user interactions is constructed by pattern recognition alone.

In this paper we describe how CERA was used to create a multimodal user interface for Skibbles, a memory game moderated by a mobile robot that interacts with humans. The Skibbles software implements CERA and demonstrates how it can be used to build multimodal interfaces. CERA and Skibbles are open-source and available as both Python and C++ code. We hope that this work practically demonstrates how multimodal interfaces can be used to develop an intelligent user interface and that our work may further this area of research.

Figure 1: Playing Skibbles. First player shows a dollar bill and then adds a Frisbee. Next player shows a dollar bill, and then mistakenly shows a floppy disk.



Playing Skibbles

Skibbles is a multi-person memory game. A mobile robot acts as the moderator, prompting the players to remember and repeat increasingly long patterns. The first player shows an object to the robot, says a word to the robot, or does both. This begins the pattern. The robot then turns to face the other player. The other player must repeat the actions in the pattern and then add an additional object, word, or both to the pattern. The robot moderator tracks the players as they attempt to repeat the growing pattern; when a player fails to repeat the pattern, he or she is "out," and the game ends. Images from a game of Skibbles can be found in Figure 1.

Figure 2: Transcript of a two-player game of Skibbles

Robot: "Welcome to Skibbles. You may begin when ready."
Player 1: Shows floppy disk, says "milk."
Robot: "You added 'floppy disk,' 'milk.'"
Robot turns to face Player 2.
Robot: "Next."
Player 2: Says "milk," shows floppy disk.
Player 2: Shows dollar bill.
Robot: "You added 'dollar bill.'"
Robot turns to face Player 1.
Robot: "Next."
Player 1: Says "milk," shows floppy disk.
Player 1: Shows dollar bill.
Player 1: Says "grits."
Robot: "You added 'grits.'"
Robot turns to face Player 2.
Robot: "Next."
Player 2: Says "milk," shows floppy disk.
Player 2: Shows dollar bill.
Player 2: Says "milk."
Robot: "You failed to complete the pattern."

Let's walk through a sample game of Skibbles. A transcript of this game can be found in Figure 2. At the beginning of the game, the first player can signal any event to the robot because there is no existing pattern. The first player shows a floppy disk and says "milk." The robot has to do visual object recognition as well as speech recognition. Because these occur within a specified time limit, Skibbles must recognize them as a single complex event. It is now the next player's turn. Skibbles must look for a floppy disk and the word "milk." According to the rules of Skibbles, these can be repeated in either order or even simultaneously, as long as both actions occur within a set time period. However, if these events do not occur within the time limit, or if another event occurs, the robot must inform the player that he has lost the game. Continuing with the game, Player 2 correctly shows the floppy disk and says "milk." He now must add another element to the pattern and chooses to add a dollar bill. The pattern that Skibbles must recognize is now more complex. First, it is composed of two events (floppy disk and "milk") that must occur within a specific duration. Second, a dollar bill must succeed this first pattern. In sum, the pattern representing each turn is embedded into a larger pattern that must be recognized in order. After several turns, the robot needs to recognize a fairly complicated pattern! Now let's skip ahead to the second to last line in the transcript. Player 2 failed to complete the pattern because he said "milk" instead of "grits." Here we see that the robot must recognize certain events as “wrong”—that is, that an event is not what is expected.

Previous Work

Our Skibbles game-playing robot demonstrates the effectiveness of using the Complex Event Recognition Architecture for creating multimodal user interfaces. The use of the Complex Event Recognition Architecture was first successfully demonstrated in monitoring a complex system for water recovery used at NASA [2]. In this system, CERA was used to build a model of event patterns occurring within the water recovery system, recognizing those event patterns in real time [5]. However, the use of CERA extends beyond the scope of the water recovery system. It can be used to describe and recognize complex patterns of discrete events [5]. Among the patterns types CERA recognizes are *One-Of, In-Order*, *All*, *Within* and *Without*. The capabilities of CERA and the patterns that it recognizes are described more fully below.

System Overview

Evolution Robot

Skibbles was built on the Evolution Robotics ER1 robot, an “off-the-shelf,” commercially available, autonomous robot that provides a software development kit and several modes of action and sensing [3]. In this project, we took advantage of ER1's speech input and output capabilities, autonomous navigation and computer vision. Evolution Robotics also provides a software application development kit, the Evolution Robotics Software Platform (ERSP[1]) [4]. The ERSP provides a suite of software modules for interacting with the robot platform, including software for low-level signaling, object detection, navigation control, and speech interaction. Because of the extensive ERSP that was simple to program, the ER1 was a good match for our project.

Python and C++ Code

Python is an interpreted, interactive, object-oriented programming language, and it is often used for application scripting [6]. C++ is a strongly-typed, non-interpreted, object-oriented programming language often used for low-level application development [7]. Because these two languages are used for widely different purposes (and, it must be said, because the ERSP provides interfaces for them), CERA and Skibbles are available as both Python and C++ code. The implementation of the software varies slightly between the two languages taking advantage of particular language constructs. However, their interfaces are remarkably similar.

[1] ERSP is a trademark of Evolution Robotics.

Integration

Building the Skibbles application was a matter of integrating the CERA software and the Evolution Robotics ERSP software to build a game-playing application. Essentially, this entailed writing code that set up the game (``Welcome to Skibbles..."), and moved the turns from one player to another. It also involved initializing the CERA Agenda, and then dynamically adding patterns to be recognized at each player's turn. During a game, when CERA informs Skibbles that the pattern was successfully recognized, Skibbles instructs the robot to recognize one additional object, word, or object/word pair. Skibbles adds this event to the existing pattern, and CERA is requested to look for this new pattern. Play moves to the next player and the pattern-matching process begins again. Skibbles is a thin system that provides for communication between the ERSP and CERA. By using the CERA system to provide the multimodal input, the actual code for Skibbles was kept quite small: fewer then 350 lines of C++ code and 200 lines of Python code.

The CERA Software

The CERA software is available as both Python and C++ code. What follows is a brief overview of the class hierarchy we developed as well as some of the details as to how we managed to incorporate the key ideas of CERA into our program.

Signals

The *Signal* class represents events that come into the CERA system from the outside world. For example, a system set up to monitor people entering and leaving a room would create a signal each time a person walked in or walked out. In the game of Skibbles, objects recognized and words spoken are represented as instances of the Signal class.

The Signal class can be extended by the user to handle his or her specific needs. All methods of the Signal class can be overridden by custom Signal subclasses to allow the user further flexibility. Creating a subclass allows the user to define the properties of the signals, such as when two signals are considered to be equal, and when one signal contravenes another (event contravention will be discussed later). In this way, the Signal class gives the user a broad range of options as to how to handle whatever types of events he or she chooses.

Patterns and Pattern Recognizers

The *Pattern* class represents patterns of signals and sub-patterns. Patterns are used to represent a larger, complex event that has taken place. Different subclasses of the Pattern class represent that one event in a specific group of events has occurred, all of the events have occurred, the events have occurred in a specific order, etc.

Associated with each Pattern class is a corresponding *Recognizer* class. The Recognizer class is the class of objects which can recognize the occurrence or non-occurrence of a Pattern. Each Pattern class has a single method, *makeRecognizer*, which creates a Recognizer object of the same type, to watch for that pattern.

Recognizer objects receive Signal instances as input. Each Signal is used to determine the current state of the Pattern instance and whether it has been fully recognized. A Recognizer is said to be *complete* if it has received the Signal instances necessary to complete the Pattern that spawned it. The specific requirements for completion depend on the type of Recognizer and underlying Pattern. If the Recognizer object receives an event that makes it impossible for the Recognizer to complete at any point in the future, the Recognizer is in a *futile* state.

The user also has the option of defining a callback function when creating a Recognizer. This function is stored internally by the Recognizer and called when the state of the Recognizer is *complete*.

What follows are brief descriptions of the Pattern classes supported by CERA.

Base Patterns

A *Base Recognizer* is a Recognizer looking for a single event. It is complete as soon as the event occurs. A *Base Pattern* is different from all other Pattern classes in that it recognizes a Signal object and not a Pattern object. Essentially, a Base Pattern is a wrapper of a Signal object. This allows all other Pattern classes to consist only of Pattern elements, and it makes recursive pattern recognition very natural.

One Patterns

One Patterns are the simplest of the Pattern types. As their name implies, they consist of exactly one element. The *One Recognizer*, the Recognizer that a One Pattern creates, is complete as soon as it recognizes that the said element has occurred.

One-of Patterns

A *One-Of Pattern* is an unordered collection of elements. A *One-Of Recognizer* is complete when any one of the elements specified by the pattern that created it occur. This allows us to extend the concept of a One Recognizer to apply to a certain group of elements.

In-order Patterns

An *In-Order Pattern* is a collection of elements in a specified order. It follows then that an *In-Order Recognizer* will be *complete* if each of the elements of the In-Order Pattern occur in the correct order. This definition is somewhat lenient in that it is acceptable for other elements to occur in between two of the elements in the pattern. Furthermore, it is possible for the elements of the pattern to occur out of order if the elements of the pattern appear multiple times. For example, if the In-Order Pattern is ABC, the pattern CBABC is acceptable. When an In-Order Pattern is *complete*, it is only guaranteed that the elements occurred in the correct order. The Recognizer ignores events that do not directly affect the next element in the list. If for some reason an event occurs that creates a condition in which the In-Order Recognizer can never *complete*, it is said to be *futile*.

All Patterns

An *All Pattern* is a collection of elements with no regard given to the order in which they occur. Quite simply, once all of the elements of a given All Pattern have occurred, an *All Recognizer* created by that pattern is *complete*.

Within Patterns

A *Within Pattern* consists of a single element, similar to a One Pattern, with the difference being that in this case a time duration is also supplied. A *Within Recognizer* can only be complete if the element that it is looking for occurs with start and finish times such that the difference between the start and finish times is less than the duration specified by the pattern.

Without Patterns

A *Without Pattern* once again consists of a single item, but in addition to the item, a time interval is specified. The Recognizer created by a Without Pattern can only be completed if the element it is looking for occurs with a start time *after* the finish time of the specified interval or a finish time *before* the start time of the specified interval. Otherwise, the Recognizer is said to be *futile*.

Allen Patterns

Allen defined the covering set of relationships that can exist between two intervals [1]. Less strictly, two intervals can be ordered just by their start times or just by their finish times, using the standard relationships $<, \leq, =, \neq, \geq, >$. For example, it may just be of interest that the start time of one interval is equal to the start time of the second interval, without regard to the ordering of their finish times. CERA provides Pattern classes and associated Recognizers for each of these Allen relationships.

The Agenda

The *Agenda* class acts as the central hub of the program. Essentially, it is a set of Recognizer objects that have been activated. The Agenda receives Signal objects asynchronously from the environment and sends each Signal to each of the Recognizer objects on the Agenda. Each Recognizer then handles the Signal according to its own specifications. Usually the Recognizer does some form of check to see if the new Signal completes its Pattern. Once a Recognizer is in a *complete* or *futile* state, it is removed from the Agenda.

An Agenda object is initialized by the user as a way to manage Recognizers and signal events to them within the scope of their program. A Recognizer is placed on and removed from the Agenda by means of the *activate* and *deactivate* methods. Generally, the user will create a Recognizer and place it on the Agenda using the *activate* method, but then allow the Agenda to determine when the Recognizer should be deactivated.

An optional feature of the *activate* method is the ability to define a *Lease* when placing a recognizer on the Agenda. This allows the user to specify a time interval throughout which the Recognizer will be eligible to receive events. The Recognizer ignores any events signaled outside the interval of its Lease.

Contravention

An important lesson we learned in writing Skibbles is that it is useful to stipulate in some patterns that we do not want to receive certain events. When we wrote the first version of Skibbles, it was only able to tell you if you successfully completed the pattern. This posed a problem because there was no way to lose the game! If you gave the robot an incorrect event, the robot would wait indefinitely for the correct response and ignore the incorrect responses. This behavior is explained by the definition of CERA's in-order Pattern class (see above). Hence, we needed the state of a pattern to be *futile* when certain events are received.

CERA's In-Order and All Pattern classes support the notion that one event could contravene another event. That is, if the next event that the Pattern is looking for has been contravened by an already seen event, then that Pattern is *futile*. For example, if we wanted a pattern that represented when only Mary, John, and Max were in a room together, it would not be correct to state that the pattern was *complete* when Mary, Sue, John and Max were in a room. Instead, we would create events for Mary, John, and Max that are contravened by any other people being present in the room. Hence, the pattern is only *complete* if all the elements of the pattern are not contravened.

Software Availability

The Python and C++ code are available on the CERA software project site at `http://www.sourceforge.net/projects/cera/`.

The software is released under an open-source license, allowing it to be freely used and modified. In providing open-source versions of the CERA software, we hope to develop a community of users that will lead to new event pattern descriptions and innovations in multimodal user interface development.

Future Work

There are two strands of future work we are pursuing. First, we would like to make CERA more accessible for developers. We hope to have a version of CERA written in the Java programming language. Java is used quite frequently in developing and prototyping multimodal user interfaces, and so this is a natural next step from a software engineering perspective. A proprietary Common Lisp version of CERA is also available from I/NET, Inc [5]. We would also like to develop an XML front-end for CERA that would allow one to easily describe the patterns for CERA to look for. This would make the pattern matching component of CERA much more natural to understand. It would also allow the robustness of CERA to be used in languages for which CERA is not natively written.

Second, we plan to develop other systems using the CERA software, both for multimodal user interfaces and for other complex event pattern recognition tasks. Any system that requires recognizing complex patterns of events, such as network intrusion detection, network system monitoring, threat assessment, and factory system monitoring, could potentially benefit from CERA. By doing this, we hope to continue to advance the understanding of complex event pattern recognition as well as provide useful software tools for developers and researchers.

Conclusions

Using CERA to provide the direct human-machine interface for a mobile robot in the Skibbles game fulfills a promise made in “Multimodal event parsing for intelligent user interfaces”—that CERA could be fruitfully and successfully used for managing *multimodal* human-machine interaction at the immediate interface level--that is, to manage asynchronous, *multimodal*, multi-channel signals [5]. As shown through the demonstration of Skibbles, CERA allows system developers to describe patterns and sub-patterns of events to be recognized by CERA in real time.

Furthermore, with this paper, we announce the availability of open-source versions of software that implement CERA, currently written in the Python and C++ programming languages. These are available at the CERA open-source software project, `http://www.sourceforge.net/projects/cera/`. Readers are invited to try the CERA software in their own projects.

References

1. Allen, J. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM* 26 *(11)*, 832–843.
2. Bonasso, P., Kortenkamp, D., & Thronesbery, C. (2003). Intelligent control of a water-recovery system: three years in the trenches. *AI Magazine 24 (1)*, 19–44.
3. Evolution Robotics. *ER1 robot description*, http://www.evolution.com/er1/.
4. Evolution Robotics. *ERSP software description*, http://www.evolution.com/product/oem/software/.
5. Fitzgerald, W., & Firby, R. J. (2003). Multimodal event parsing for intelligent user interfaces. *Proceedings of the 2003 International Conference on Intelligent User Interfaces,* ACM Press, 253–261.
6. Python Software Foundation. *The Python Programming Language*, http://www.python.org.
7. Stroustrup, B. (1997). *The C++ Programming Language*, third ed., Addison-Wesley.

Acknowledgements

CERA was originally developed under NASA SBIR Projects 99-1-08.04-1856 and 99-2-08.04-1856. The Python and C++ code was developed a 2003 Summer NSF REU Program at Hope College.