The Monitoring and Notification Flow Language

Alex Edgcomb and Frank Vahid
Dept. of Computer Science and Engineering
Univ. of California, Riverside
aedgcomb/vahid@cs.ucr.edu

ABSTRACT

Assistive monitoring analyzes data from sensors and cameras to detect situations of interest, and notifies appropriate persons in response. Customization of assistive technology by end-users is necessary for technology adoption and retention. We introduce MNFL, the Monitoring and Notification Flow Language, developed over the past several years to allow lay persons without programming experience, but with some technical acumen, to effectively program customized monitoring and notification systems. MNFL is a graphical flow language having intuitive yet sufficiently powerful execution semantics and built-in constructs targeted to assistive monitoring. We describe the language's semantics and built-in blocks, demonstrate the language's use for customizing several common assistive monitoring tasks, and provide results of initial usability trials showing that lay users with almost no training on MNFL can more than 50% of the time and in just a few minutes select and connect the right 1-2 blocks to complete applications that have 4-5 blocks total.

Keywords

Assistive monitoring, smart homes, telehealth, telemonitoring, end-user programming, monitoring, notification, sensors, embedded systems, ubiquitous systems.

1. Introduction

Assistive monitoring uses technology to monitor a person's activities (typically in the person's residence) for situations of interest, and to respond with automated notifications. Examples include monitoring for a sound like a doorbell ring and responding with visual cues like blinking lights to assist a deaf person, or monitoring for an elderly live-alone person arising in the morning and responding by sending a text message to a caregiver. The continued decreasing costs of cameras, sensors, computers, and broadband connectivity have led to recent proliferation of assistive monitoring systems.

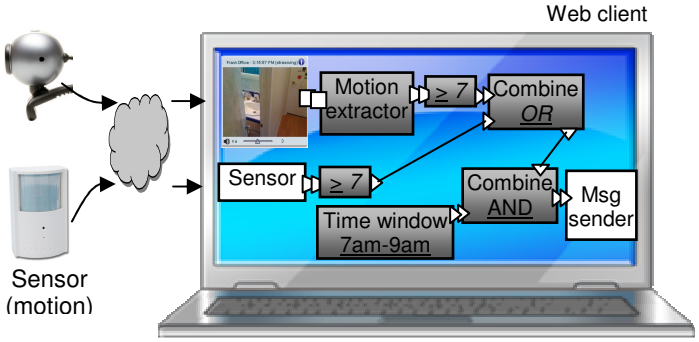
Assistive monitoring systems are commonly installed or provided as a kit by a vendor. A vendor may provide a kit consisting of several sensors, audio/video cameras, and a basestation that streams sensor/camera data to a local computer or remote compute server. The end-user might place the sensors and cameras throughout a home to monitor an aging parent, or in a unit of an assisted living facility to monitor an at-risk resident. The term *end-user* refers to the people that configure and maintain the system, such as the adult children of an aging parent or the IT staff of an assisted living facility, and not necessarily to the people being monitored. *The ability of the end-user to customize an assistive monitoring system is essential to the system's practical use.* Lack of customizability is a primary inhibitor of assistive technologies. Users not involved in the initial application of assistive technology usually abandon the technology within a year [10][26]. Users not able to modify the technology to changing needs or priorities are also likely to abandon the technology [20][25].

Existing end-user programming methods do not provide the necessary expressivity or ease-of-use required for adequate end-user programming of assistive systems. This paper introduces MNFL, the Monitoring and Notification Flow Language, a new block-based graphical flow language (illustrated in Figure 1) intended to enable end-users to effectively program customized monitoring and notifications for a variety of assistive monitoring applications. By focusing specifically on assistive monitoring when creating the language, the language provides sufficient expressivity while being usable by end-users. The paper summarizes previous end-user programming efforts, introduces MNFL syntax, semantics, and built-in constructs, provides a series of examples illustrating MNFL's use for programming common assistive monitoring applications, shows how MNFL's block-based approach gives end-users various customization options, and describes usability trials demonstrating that lay people can successfully program using MNFL.

2. End-user programming

Commercial assistive monitoring systems typically allow end-users to select from a pre-defined menu of monitoring and notification options, resulting in limited customizability. For example, Motorola's Homesite system allows end-users to associate an event, such as taking of a picture, recording of audio/video, or an e-mail notification, with the triggering of a sensor [19]. ADT's GrandCare system [11] relays a home's sensors to a central server that analyzes sensor data to determine usual behavior for each sensor. An end-user programs GrandCare by configuring notifications to particular situations, such as pills not taken by 11:00 a.m. or unusual door activity. GE's QuietCare system [21] and the WellAware system [32] have motion sensors and contact switches connected to a basestation, which relays sensor information to a central server. The server learns typical behavior patterns, and caregivers are alerted of variations from those patterns. The end-user can configure the alert form but not the alert cause. WellAware also maintains a history for each sensor that can be viewed by the end-user at a website, analogous in appearance to a car's monitoring gauges.

Figure 1: Sample assistive monitoring system that monitors for motion via a camera or a motion sensor and responds by sending a text message.



Alternatively, an end-user can build a monitoring and notification system with pre-made kits from a variety of online stores [12][16][17][27]. For example, SmartHome offers a variety of monitoring kits and products, such as an eight camera with basestation security system and a programmable thermostat [27]. Similarly, Harris Communications [12] has thousands of individual products for deaf assistance, such as a bed-shaker that responds to a ringing phone. However, such kits and products do not interoperate, leading to excessive devices and chaotic uncoordinated notifications. Additionally, installing the components may require an electrician and programming the components may require writing custom applications.

Numerous research efforts have sought to teach people without programming experience and who are not actively training to become programmers (referred to in this paper as "lay people") to learn sequential programming, the main form of desktop computer programming wherein programmed instructions execute in sequence. For example, Logo is a sequential programming language best known for the turtle [24]. The turtle represents the head of a pen, which is controlled by instructions, such as move pen forward 100 steps and turn pen 90 degrees. Logo also supports loops and recursion. Scratch is another sequential programming language that uses colored shapes to indicate functionality and purpose [23]. Shapes that visibly appear to piece together represent legal combinations of the shapes in Scratch. Another example is Lego Mindstorms, which is predominantly used to program robots [15][31]. Mindstorms uses functionality blocks, such as a stopwatch, comparator, and ultrasonic sensor, which can be drag-and-dropped onto a graphical timeline. The goal of introducing people to sequential programming and more recently "computational thinking" is much grander than the goal of programming assistive monitoring systems, and has a longer steeper learning curve.

Context-aware applications adapt their behavior to their environment, such as a mobile tour guide device application that plays audio segments based on the devices current location in a museum [7]. Researchers have sought to enable lay end-users to customize context-aware applications. The a CAPpella approach [6] allows users to demonstrate desired behavior. The system records an application's inputs and behavior while the user manually carries out the desired behavior, annotating the relevant portions of the recording for the system. The system's automated machine learning techniques then strive to learn the behavior, such as "Whenever the front-door-sensor detects motion, then the front-door-camera should record for 10 seconds." Machine learning is also used in the adaptable smart-home technology of CASAS [22], wherein sensors are monitored to detect a user's normal activity patterns, and automation policies are learned that mimic these patterns. Anomalies from normal patterns can also be detected, as in some commercial systems like GE's QuietCare [21]. CAMP [29] allows users to describe desired context-aware behavior, by stringing together a set of available words, akin to popular refrigerator magnets with words that people string together to form sentences (aka "magnetic poetry"). An example might be "When front-door-sensor motion then front-door-camera record 10 seconds." The system parses these sentences and strives to comprehend the meaning, which is an easier task than understanding general language sentences due to the limited pre-defined vocabulary. A multimedia editing tool approach, called MAPS, allows users to program scripts using a filmstrip-like view, wherein users can compose or modify a sequence of pictures, audio clips, and annotations [3].

Our MNFL approach focuses specifically on assistive monitoring, seeking to provide a powerful direct programming environment to end-users while not requiring programming expertise. Generally, our approach is complementary with previous approaches—a system may provide a variety of programming mechanisms, including learning from demonstration or from normal activity patterns, description through magnetic poetry or filmstrip approaches, our graphical flow-language approach, as well as more traditional textual or graphical sequential or state machine programming for more advanced users.

3. MNFL

This section introduces the Monitoring and Notification Flow Langauge—MNFL. Note that this introduction targets technical experts, and is not the introduction that would be provided to end-users, who instead would be introduced through usage examples.

3.1 Camera/sensor framework

For concreteness, we describe the ViaSee [30] client/server framework into which MNFL is currently integrated. However, MNFL can be integrated into any framework having audio/video and sensor stream inputs and having notification capabilities.

The ViaSee framework enables setup and sharing of camera and sensor streams. An in-home personal computer (Windows or

Figure 2: Existing framework enables live monitoring of accessible (owned or shared) camera and sensor streams via a web client.

Mac) runs the ViaSee Streaming Software application, which detects standard webcams or eBlock sensors [4][5] attached to the PC via USB. A user can authorize specific webcams and sensors for streaming to the user's ViaSee web account, such streams are said to be "owned" by the user. When logged into the web interface, the user can view all owned streams. Furthermore, the user can share any stream with other ViaSee accounts, via a Facebook-friends-like invitation scheme. A user can instantiate, place, and size any accessible streams (owned by or shared with the user) into one or more web windows. A sample window in the web client is shown in Figure 2.

3.2 Language overview

MNFL is a graphical flow language, wherein a user instantiates and connects predefined blocks to define desired behavior. Figure 1 provides an introductory example. The system has camera and sensor input streams. The MNFL program's behavior is to send a text message if motion greater than or equal to 7 is detected by either input, and the time is between 7:00 a.m. and 9:00 a.m. This example will be explained in more detail later.

Each block begins executing immediately upon being instantiated, repeatedly reading the block's input stream, performing some computation, updating an internal state, and generating an output stream. Such "always executing" functionality, coupled with data flowing from block outputs to other blocks' inputs, is intuitive to lay people due to similarity with the physical world. In contrast, some graphical programming languages use interconnected blocks that then execute in sequence and that transform variable values, where such sequencing and variables have few physical analogs and are thus less intuitive to lay people. Flow languages have long been shown to be usable by lay people [13][18][28]. MNFL's pre-defined blocks will be further described in Section 3.5.

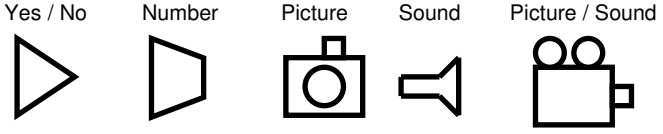
MNFL's ***data value types*** include:

- *Integers*: Positive or negative whole numbers. The currently supported range is -9,999 to 9,999 (suitable for most assistive monitoring applications). This type is called a "number" type for easy comprehension by lay people.
- *Booleans*: "Yes" and "no" values. This type is called a "yes/no" type for easy comprehension by lay people.

The data value types are used almost exclusively for inputs and outputs of blocks (MNFL does not possess "variables"). In Figure 1, inputs/outputs of integer types are graphically depicted as trapezoids, and yes/no types are depicted as triangles. For example, the "Sensor" block has an integer output. The ">= 7" block (known as a "Threshold" block) has an integer input, and a yes/no output that will be "yes" when the input integer is greater than or equal to "7," and "no" otherwise. Because MNFL deals with audio/video too, two additional data value types are included:

- *Sound:* An instance of audio. This data value type seems a bit unusual, but is defined for consistency, as will become clearer below.
- *Picture:* A 2-dimensional color image.

Figure 3: Depictions of the four core input/output data value types: yes/no, number, picture, and sound. Also depicted is the common picture+sound data value type.



Picture inputs/outputs are depicted as a picture camera, and sound as a speaker, as shown in Figure 3. Because audio/video cameras, such as web cameras, typically have a built-in microphone, we also include a picture+sound data value type, shown as the movie camera in Figure 3.

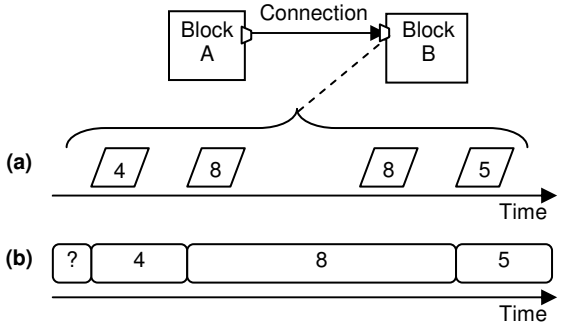
Because MNFL deals with data over time, the language also defines three ***data duration types***—a concept that doesn't exist in typical programming languages, and the reason why the previous paragraph used the term "data value types" rather than the more common simpler term "data types". A *snapshot* is a single data value at a given time instant. In contrast, a *stream* is a continuous sequence of data values conceptually extending forever into the future. A *snippet* is a portion of a stream with a beginning and end. An example of a stream is a camera block outputting a stream of pictures, which people commonly call "video." A microphone may output a stream of sounds, commonly called "audio." Likewise, a motion sensor block continuously outputs a stream of integers (for which a common name does not exist). Sampling a picture stream at an instant in time yields a snapshot, i.e., a single picture. Likewise, sampling an integer stream at an instant in time yields a snapshot, i.e., a single integer. Recording the picture or integer streams for X seconds results in X-second snippets. A block's input or output is usually a stream. Snapshots and snippets are special cases, usually associated with a block's internal functionality.

MNFL includes a simple casting of data value types between yes/no and integer types. Going from yes/no to integer, a yes is cast to a 1 and a no to a 0. In the other direction, a non-zero integer is cast to a yes, and a 0 is cast to a no.

3.3 Continuous-time I/O abstraction

Conceptually, the time separation between one value and the next in a stream is infinitely small. However, practical implementations use "small enough" separations such as 30 pictures per second for a camera or 5 integers per second for a motion sensor. Furthermore, such values are typically communicated in discrete chunks known as packets. MNFL uses a continuous stream abstraction of such discrete packets, which involves interpreting a series of received values in such packets as a continuous stream. Figure 4 demonstrates how a series of integer packets are interpreted as a continuous stream by the receiving block. In short, a value received in a packet is extended until replaced by a new packet's value. Note that the number of times that the same value is repeatedly sent is irrelevant and only packets with a different value from the previous packet matter in the

Figure 4: Continuous integer-stream abstraction in MNFL: (a) received packets, (b) continuous interpretation.



interpretation. (In practice, a physical implementation protocol may require that packets be sent at a minimum rate, else the receiver can assume the sender is faulty or disconnected, but that subject is beyond this paper's scope).

Furthermore, in any practical implementation, non-zero compute time occurs between the time one block outputs a value and the time the next block receives and processes that value. The abstraction simply changes the value after processing; no attempt is made to time-stamp values or to impose zero-delay semantics on the language (as for example in languages like Esterel [1][2]). Due to the nature of the domain of monitoring and notification, such an abstraction in MNFL does not lead to problems; the same could not be said if MNFL were used in some other domains like real-time control or real-time signal processing.

References in this paper to a "continuous integer-stream" or similar are understood to refer to the above-described abstraction.

3.4 Block execution semantics

MNFL's execution semantics for a connection of blocks is a variation of Kahn processing networks (KPN) [14]. In KPN, processes communicate via channels. A process executes when all of the process' inputs have a token, and generates output tokens after such execution. In MNFL, each block corresponds to a KPN process, and each connection corresponds to a KPN channel. However, a block executes when any (rather than all) of the block's input connections have a token. Additionally, MNFL prohibits block communication loops.

Each block has a vector of N-inputs *I* and M-outputs *O*, where N and M are a positive integer or 0. Each input has at most one connection, and each output has an unlimited number of connections. *I* passes tokens to queue *Q* of unprocessed tokens. Inside the block is a vector of N-previous inputs *P* The state machine in Figure 5 describes how tokens are processed. While *Q* is not empty, an unprocessed token from input *I[i]* is popped from *Q*. If the token is not the same value as the previous token of the same input *P[i]*, then a function *f()* converts *P* to *O*. Otherwise, the state machine does nothing with the token.

3.5 Built-in blocks

Figure 5: State machine of each block. Each block has a queue *Q* of unprocessed block inputs. *I* is a vector of block input values. *P* is a vector of past block input values. *O* is a vector of block output values. *f()* is a function that converts the block inputs into the block outputs.

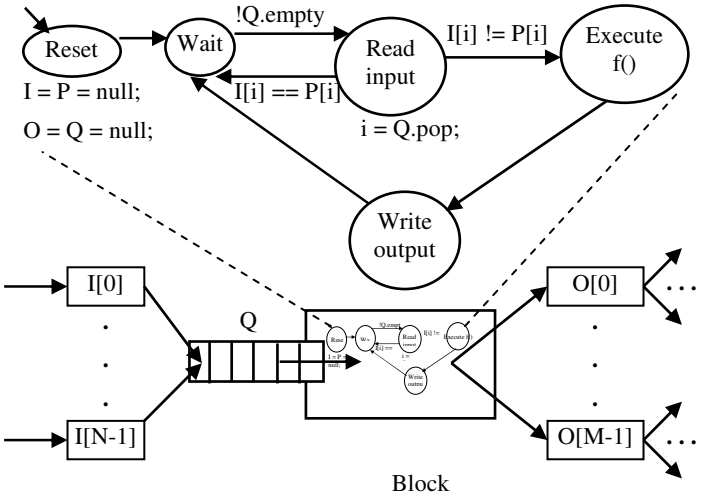
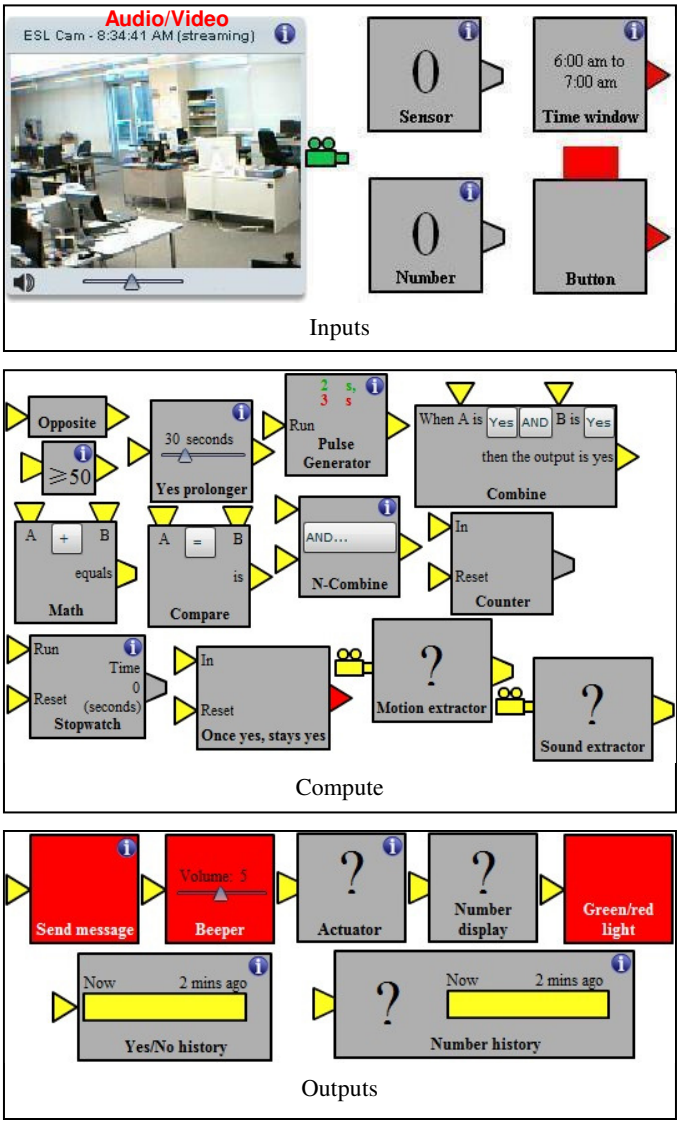


Figure 6: Built-in MNFL blocks.



MNFL has built-in blocks targeted to assistive monitoring, forming the language's core constructs. A block can have inputs or outputs. Block inputs appear on the block's left, while outputs appear on the right. A yes/no input or output triangle is red when the input/output data value is no, green when yes, and yellow when the data is "error." The semantics of errors will be discussed in Section 3.6. A number input or output trapezoid is grey when the data is a number, and yellow if the data is "error" or if no data is present. A user placing a cursor over the trapezoid causes the current number to appear. A picture, sound, or picture+sound input or output is green when data is present and yellow otherwise.

A block with no inputs and at least 1 output is an *input block*. A block with no outputs and at least 1 input is an *output block*. A block with both inputs and outputs is a *compute block*. The block's are shown in Figure 6. Some blocks have an "i" icon in the top right corner. Clicking the icon opens the block's configuration window.

The key challenge in creating MNFL's built-in blocks was to define the *fewest* blocks possible, so that users can find needed blocks in the list of built-in blocks, while defining *intuitive* blocks sufficient to achieve assistive monitoring goals without requiring

excessive numbers of blocks to be connected.

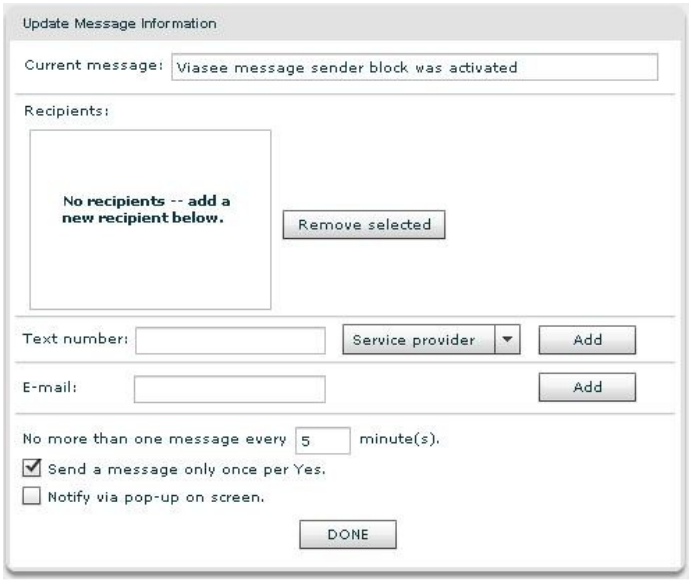
MNFL's built-in input blocks are:

- Audio/Video—Associated with a physical camera, outputs audio and video. The configuration window adds/removes other accounts from accessing the stream. Additionally, the association between the physical camera and block can be broken from the configuration window.
- Sensor—Associated with a physical sensor, outputs a yes/no or number stream. The sensor block displays the current data on the block. The configuration window has the same options as the audio/video block's configuration window.
- Time window—Outputs yes during a user-configurable start time and end time. Otherwise, outputs no. By default, all days are set to have the time window, but a user can specify particular days. The configuration window allows the user to change the start/end times and specific days of time window usage. The default time window is 6am to 7am everyday of the week.
- Number—Outputs a user-defined constant number. The configuration window allows the user to change the number. The default number is 0.
- Button—Outputs a user-defined constant yes/no. Clicking the red rectangle (using a mouse) changes the rectangle to green and causes the block to output yes. The default rectangle is red and output is no.

MNFL's built-in compute blocks are:

- Opposite—Converts an input yes to an output no, and vice-versa.
- Threshold—Outputs a yes if the input number equals or exceeds a user-defined number. The configuration window allows the user to change the user-defined number. The default number is 50.
- Yes prolonger—Extends a yes input into a longer yes output, e.g., a 2 second input yes may be extended by 8 seconds at the output, resulting in a 10 second output yes. The length of extension time is adjustable from a slider on the block, which ranges from 0 to 99 and is changed by clicking on the slider bar. The configuration window allows the user to change the time granularity. The time granularities are: 1/10th seconds, seconds, minutes, hours, days. The default extension time is 1 and the default granularity is seconds.
- Pulse generator—When activated via a yes at the block's input, outputs yes for a user-defined length of time, then outputs a no for another user-defined length of time, and repeats, thus pulsing the block's output. When the block input is no, the output is no. The configuration window allows the user to change the yes and no times of each pulse. The time granularities for each of yes and no are user configurable. The default yes length is 1 second, and the default no length is 1 second.
- Combine – Outputs a yes/no value based on a user-defined logic operation applied to two yes/no inputs. The sentence on the block, "When A is Yes AND B is Yes then the output is yes" has three configurable segments, each a drop-down list. The first "Yes" can be either "Yes" or "No." The "AND" can be either "AND" or "OR." The second "Yes" can be either "Yes" or "No." The default configuration is "Yes", "AND", and "Yes." Earlier work showed that logic is hard for lay users, and the sentence block had most success [5].
- Math—Outputs the number result of a user-defined arithmetic operation on two number inputs. The arithmetic operations are configured through a drop-down list with options "+", "-", "*", and "/". The default operation is "+".
- Compare—Outputs the yes/no result of a user-defined comparison operation between two yes/no inputs. The comparison operation is configured through a drop-down list with options "=", "≥", ">", "≤", "<", and "≠". The default operation is "=". This block is a more advanced form of the more common Threshold block.
- N-Combine—Outputs the result of a user-defined logic operation on a user-defined number of yes/no inputs. The logic operation is configured from a drop-down list on the block with options AND, OR, XOR, NAND, NOR, and XNOR. The number of yes/no inputs is configured from the configuration window with a range of 2 to 8 inputs. The default operation is AND. The default number of inputs is 2. This block is a more advanced form of the more intuitive (for lay people) Combine block.
- Counter—Outputs the number of rising edges (no-to-yes transitions) counted on the block's "In" input. The count is cleared to 0 when the "Reset" input is yes, which has priority. The default count is 0.
- Stopwatch—Outputs the amount of time the "Run" input has been yes. The configuration window allows the user to change the time granularity of the output. The time is cleared to 0 when the "Reset" input is yes, which has priority. The default output time is 0.
- Once yes, stays yes—Outputs a yes when the "In" input becomes yes, continuing to output yes until the "Reset" input becomes yes, which has priority. The reset causes the output to become no. The default output is no. This block is commonly called a Tripper by engineers.
- Motion extractor—Outputs the amount of motion as a number based on the video input.

Figure 7: The configuration window for the send message block is where the user can modify the message sent, list of recipients, and method(s) of notification.



- Sound volume extractor—Outputs the intensity of sound as a number based on the audio input.

MNFL's built-in output blocks are:

- Send message—When the block's input is yes, a user-defined message is sent. The configuration window, shown in Figure 7, has a message field, a recipient list that contains all recipients to receive the message, a minimum duration in minutes between sending messages field, a one message per yes checkbox, and a pop-up message enabling checkbox. A recipient can be input as an e-mail address or a phone number for text message. The default message is, "Viasee message sender block was activated." The default recipient list is empty. The default duration between messages is 5 minutes. By default, the message is sent once per yes, and the pop-up window is unchecked.
- Beeper—When the input is yes, the block uses the computer on which MNFL is running to output an audible beeping sound. Otherwise, the beeping is off. A slider on the block controls the volume of the beeper and has the range of 0 to 10.
- Actuator—Associated with a physical actuator, which expects a yes/no or number input. The association can be broken in the configuration window. The current input value is displayed on the block itself as a number.
- Number display—Shows the current input value as a number on the block itself.
- Green/red light—Shows the current yes/no input value changing the entire block's color to green or red.
- Yes/No history—Records the input as a yes/no value paired with a time stamp when the input value changes. Displayed on the block is a graphical window showing the recorded history in time slices that display the average of the recorded values (yellow for error, green for yes, and red for no). Significant outlier values and error values supersede the average values. The most recent time slice is on the left-hand side of the window. The length of time represented by the time window can be changed in the configuration window. Also in the configuration window, the user can save a comma separated value (CSV) formatted file with the following columns: date, time, and value. Lastly, in the configuration window the user can specify how long to retain the recorded data. By default the time window is set to 2 minutes and the data retention is set to no time limited.
- Number history— Records the input as a number value paired with a time stamp when the input value changes. The displayed history is a two dimensional graph, analogous to a stock market graph. Significant outlier values and error values supersede the average values. The current input value is shown as a number to the left of the graph. The length of time represented by the time window can be changed in the configuration window. The configuration window includes a user-specifiable maximum and minimum scale value to be displayed in the time window. Also in the configuration window, the user can save a comma separated value (CSV) formatted file with the following columns: date, time, and value. Lastly, in the configuration window the user can specify how long to retain the recorded data. By default the time window is set to 2 minutes, the data retention is set to no limit, the maximum scale value is 100 and the minimum is 0.

As with most languages, additional libraries that include more advanced blocks, but the above blocks come built-in with MNFL.

3.6 Error handling

Section 3.2 indicated that blocks begin executing immediately upon instantiation. A normal situation is that a recently-instantiated block begins executing before the user has connected the blocks inputs to other blocks' outputs. As such, block behavior must be defined to handle unconnected inputs. MNFL thus includes a special "error" value for all the language's data value types. For example, a yes/no input may have three possible values: yes, no, or error. A number input may have an integer value (e.g., -5, 0, 9, …) or the value "error." A picture input may be a picture or the value "error." "Error" values also commonly occur when a physical camera or sensor is temporarily disconnected (such as when being moved) or temporarily fails (due to a failed battery for example).

Due to the commonness of error inputs, block behavior is defined to respond gracefully to error inputs. For example, recall that a stopwatch block increments the block's output value (at the appropriate time interval) when the block's "Run" input is "yes." The block is defined such that the block's output value is 0 upon instantiation. If the "Run" input has an "error" value, the block simply doesn't count, and does *not* reset the output or output an error value. Thus, if an application for example is counting the time that motion is sensed, and the motion sensor fails, the counted time is kept, with counting resuming once the motion sensor resumes operating correctly. As another example, a combine block configured for OR will output a "yes" if any input is "yes," even if some inputs have error values, because those inputs don't matter if the correct output can be determined from non-error inputs. If all non-error inputs are "no," then the block outputs "error" because the correct output value can't be determined.

3.7 Feature extractors

Monitoring requires a means for combining camera and sensor streams. The blocks in Section 0 support user programming involving number and yes/no data value types. Picture and sound types can be incorporated for purposes of assistive monitoring by introducing another category of blocks known as extractors. An ***extractor*** block has a sound or picture stream (audio or video, respectively) as input, and outputs a stream of integers representing the value of a particular feature found in the stream. An example is a sound volume extractor that outputs the intensity of sound in input audio, or a motion extractor block that outputs the amount of motion in an input video. Extractors typically output a number between 0 and 100, 0 meaning the feature is not detected at all (no motion, or no sound), and 100 meaning the maximum amount is detected (the detection is saturated).

Extraction is a flexible mechanism for satisfying many user wishes regarding monitoring audio and video. For example, a major concern with audio and video is privacy. Since feature extractors obfuscate the input by converting the input into numbers, privacy can be increased by placing the feature extractor closer to the camera. Additionally, features extractors are capable of detecting important monitoring-related events that typical sensors cannot, such as fall and scream detection.

4. Examples

This section provides several examples of end-user programming of common assistive monitoring applications using MNFL.

4.1 Person at door alert for hard-of-hearing

A common goal for the hearing impaired is to be notified when someone is at the door. In Figure 8, a camera is placed inside a front door. The camera is instantiated in MNFL as an audio/video block and connected to a doorbell extractor block that outputs the likelihood that a doorbell ring has been detected, as a number between 0 and 100. The doorbell extractor block is connected to a threshold block, set to a threshold of 50 but adjustable by the user based on usage experience (i.e., desired sensitivity). The threshold block is connected to three actuator blocks that are individually associated with the living room lights, front room LEDs, and a bed shaker. The threshold block is also connected to a send message block configured to send a text message to the hearing impaired person's cell phone (which hearing impaired people commonly carry) with the message "Doorbell rung," sent at most once per minute.

Because a person might knock instead of ring the doorbell, Figure 9(a) adds a knock extractor block, connected to a threshold block. The threshold block from the knock extractor is connected, along with the threshold block from the doorbell extractor, to an n-combine block set to OR-logic. Therefore, the n-combine block outputs yes when a person is at the front door. The n-combine block could then connect to the three actuator blocks and the send message block.

The hearing impaired person may desire the capability to enable/disable the system with the flip of a switch. Figure 9(b) features a sensor block associated with an enable switch. The enable switch block is connected, along with the earlier n-combine block detecting a doorbell or knock, to an N-Combine block configured for AND-logic. The AND n-combine block outputs yes when both the enable switch is yes and a person is at the door. The block's output can connect to the three actuator blocks and the send message block.

The multiple forms of notification can be further improved by pulsing the actuations. Figure 9(c) introduces a yes prolonger block set for 20 seconds, connecting to a pulse generator block configured for 2 seconds of yes and 2 seconds of no. The pulse generator block connects to the three actuator blocks and the send message block. When a person is at the door and the enable switch is yes, the living room lights, front room LEDs, and bed shaker will for 20 seconds pulse on/off. Only a single text message is sent because the send message block is configured to send at most one message per minute.

While the network in Figure 9 may look slightly imposing, we reiterate that an end-user would likely extend functionality incrementally as above, which is made easy due to the always-execute block structure of MNFL. Note also that, during trial-and-error development, the extractor blocks could be replaced by number blocks, and the end-user could then type a number to replicate a doorbell or knock and to view the network's behavior.

Figure 8: Person-at-front-door notification system for a hearing-impaired person.

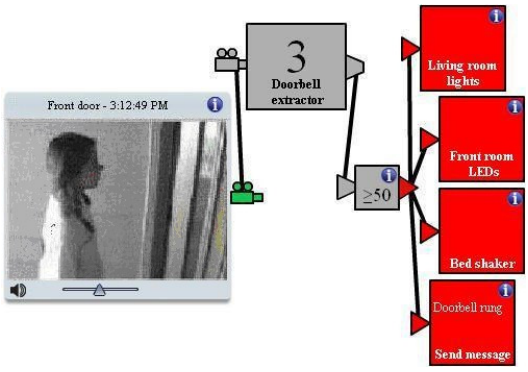
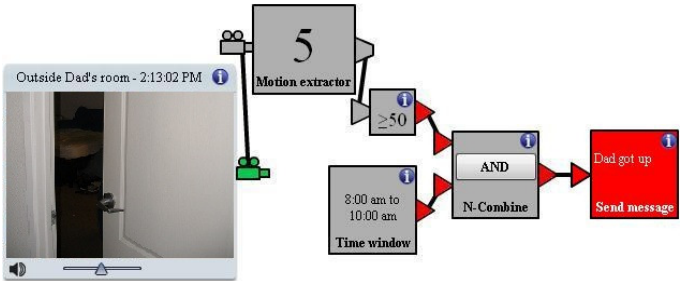


Figure 10: Motion outside the room sends one text message in the morning to the caregiver indicating that, "Dad got up."



4.2 Morning rise notification for caregiver

Ensuring a person arises in the morning is a common goal for caregivers. A first solution version sends a text message when the person ("Dad") leaves the bedroom during his normal rising time between 8 am and 10 am. A solution is shown in Figure 10 using a camera outside the bedroom. The camera is associated with the audio/video block, which is connected to a motion extractor block and then threshold block set to 50. The threshold block is connected to an n-combine block using AND-logic along with a time window block configured to output yes between 8 am and 10 am. The n-combine block is connected to a send message block configured to send only once per 120 minutes and configured to send the message "Dad got up."

The next version uses one of three events to cause sending a text message to the caregiver when Dad rises in the morning: 1) motion outside the bedroom door, as before, 2) significant motion inside the bedroom, 3) a button press in which the button is located near the bed. In this variation, morning is defined as between 7 am and 9 am. The solution in Figure 11 uses a camera inside the bedroom, a camera outside the bedroom, and a button near the bed. The camera audio/video blocks connect to motion extractor blocks, which connect to threshold blocks. The threshold block stemming from the camera inside the bedroom has a higher threshold of 80 to detect rising versus just normal in-bed motion. Both threshold blocks connect to an n-combine block with OR-logic, along with a sensor block associated with the button near the bed. This 3-input n-combine block connects to another n-combine block using AND-logic, also connected to a time window block configured to output yes between 7 am and 9

Figure 9: Three customizations of the person-at-front-door system, shown all at once for space reasons: (a) Detecting knocks in addition to the doorbell, (b) including a physical switch for system enable, and (c) pulsing the actuators for 20 seconds for better notification.

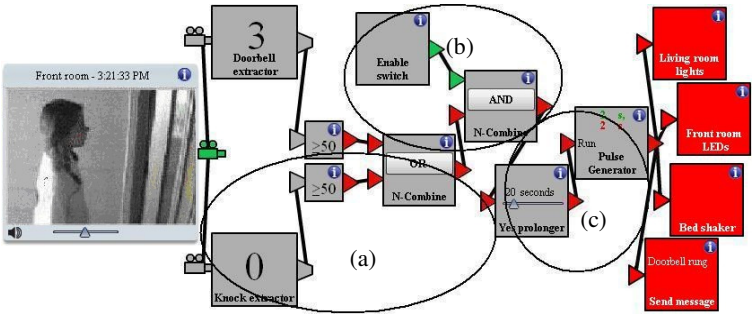
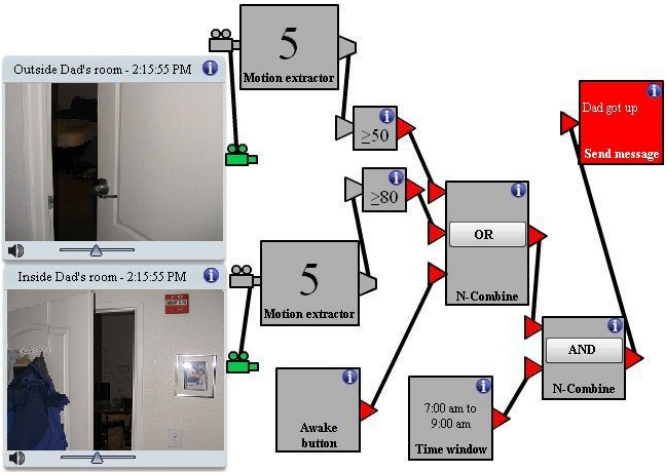


Figure 11: Motion outside the bedroom, significant motion inside the bedroom, or a button press sends one text message in the morning.



am. This n-combine block connects to a send message block configured to send a message no more than once every 120 minutes with the message "Dad got up."

The third version sends a text message if Dad has *not* arisen by 10 am, sending no notification otherwise. Also, Dad wants the ability to enable/disable the system with a switch. A camera is mounted inside and another outside the bedroom. In the solution in Figure 12, both associated audio/video blocks connect to motion extractor blocks and threshold blocks, as before. Both threshold blocks connect to an n-combine block using OR-logic, which connects to the "In" input of a once-yes-stays-yes block. A time window block configured to output yes between 8 am and 10 am is connected to an opposite block that is connected to the "Reset" input of the once-yes-stays-yes block. Therefore, the once-yes-stays-yes block will output yes only if there has been sufficient motion inside or outside the bedroom between the hours of 8 am and 10 am. The once-yes-stays-yes block is connected through an opposite block (indicating *no* motion between 8 am and 10 am) that in turn connects to an n-combine block with AND-logic. Also connected to the n-combine block is a sensor block associated with the enable/disable switch, and a time window block configured to be yes between 10:00 am and 10:01 am. This n-combine block connects to a send message block configured to send the message "Dad did not get up."

4.3 In room too long notification for caregiver

A common situation of concern for caregivers is a person being in a particular room, typically a bathroom, too long, suggesting a possible fall or other need for assistance. The solution in Figure 13 uses a camera mounted in a hallway outside a bathroom, facing perpendicular to the bathroom door. The associated audio/video block connects to a rightward motion extractor block and a leftward motion extractor block, which connect to threshold blocks set to 50. The rightward threshold block connects to the "In" input on a once-yes-stays-yes block, which thus outputs yes when a person enters the bathroom. The leftward threshold block connects to the "Reset" input of the once-yes-stays-yes block, thus resetting the block to no when a person leaves the bathroom. The once-yes-stays-yes block is connected to the "Run" input of a stopwatch block, configured to output minutes, such that the stopwatch counts the time a person is in the bathroom. The leftward threshold block also connects to the "Reset" input of the stopwatch block to clear the count when a person leaves the bathroom. The stopwatch block connects to a threshold block

Figure 12: If no motion occurs outside the bedroom and no significant motion occurs inside the bedroom in the morning, then a text message is sent at 10 am. This system can be disabled by a physical switch.

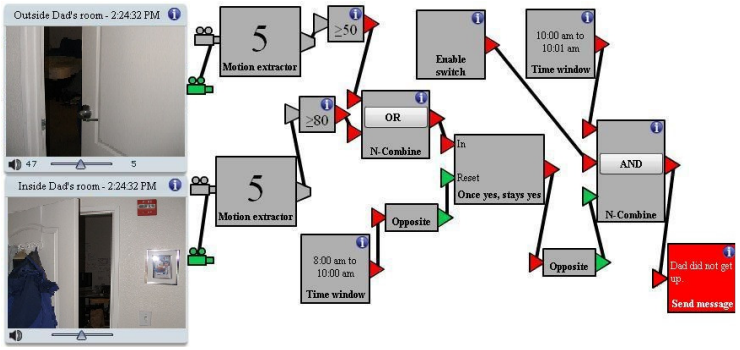


Figure 13: When a person enters the bathroom without leaving for 60 minutes, a text message is sent to the caregiver.

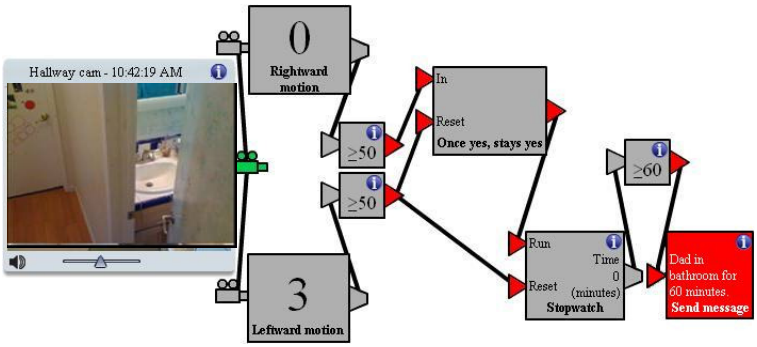
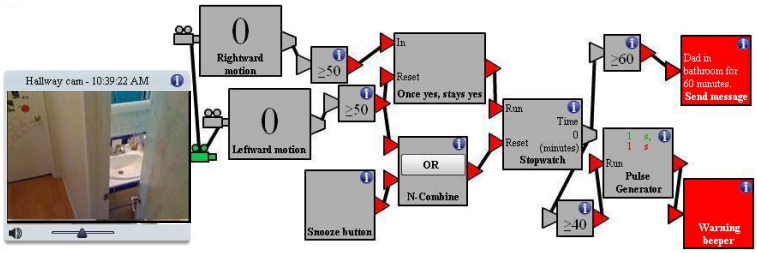


Figure 14: A snooze button resets the stopwatch for the person in bathroom too long design from Figure 13. Also, a warning beeper sounds 20 minutes before the message is sent.



configured to 60, which connects to a send message block configured to send a text message to the caregiver with the message "Dad in bathroom for 60 minutes."

Perhaps Dad would like more flexibility via a button that adds another hour before the text message is sent (akin to a "snooze" button on an alarm clock). Also, a warning beeper is added to the bathroom that gives Dad a 20 minute warning before the message is sent (at which point he may choose to push the button). The solution in Figure 14 extends the previous solution accordingly. Instead of the leftward threshold block connecting directly into the "Reset" input of the stopwatch block, the leftward threshold block connects to an n-combine block with OR-logic, which also receives input from a sensor block associated with the snooze button. The stopwatch block now also connects to a second threshold block configured to 40, which connects to a pulse generator with a 1 second yes and 1 second no. The pulse generator block connects to an actuator block associated with the beeper.

4.4 Discussion

The above examples illustrate MNFL's ability to capture common assistive monitoring goals merely via instantiation and connection of built-in blocks, and also demonstrates the ability of end-users to customize systems to their particular requirements.

Furthermore, MNFL seamlessly crosses between physical and virtual environments because of the always-execute block-based flow approach of the programming language. For example, a physical motion sensor in the living room and a physical motion sensor in the bedroom may be connected to a *physical* combine block with OR-logic. The physical combine block may then be connected to a basestation that streams yes/no values to a computer. Alternatively, the physical motion sensors may connect directly to the basestation and stream numbers to a computer, where the OR can be done using an MNFL n-combine block.

5. Usability trials

We conducted trials to determine the usability of MNFL by end-users. The term end-user does not necessarily refer to the person being monitored, but to people that configure and maintain the system, such as the IT staff of an assisted living facility or the adult children of an aging parent. Although MNFL end-users are not assumed to have programming skills, end-users are assumed to have basic technical competence, similar to the competence required to set up a home wireless network or to synch online and mobile phone calendars.

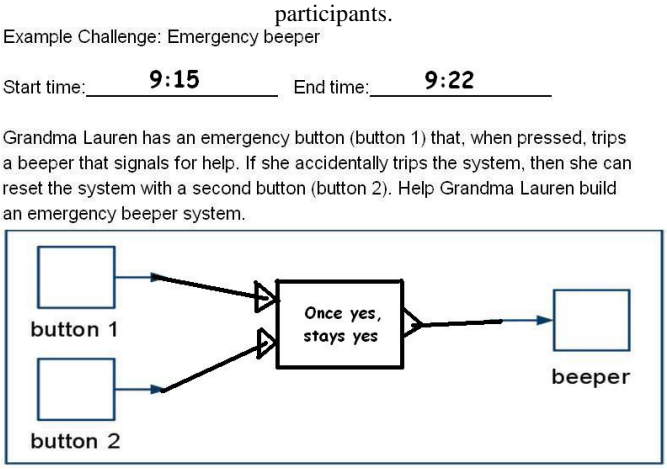
5.1 Participants

The participants were non-engineering undergraduate students at the University of California, Riverside (UCR). The participants were enrolled in CS 8, a course that introduces Microsoft Office and HTML, intended for non-computing and non-engineering majors and having no prerequisites. Most UCR degree programs outside science and engineering schools require undergraduates to take any course in computing or math as an elective; CS 8 attracts approximately 2,000 students per year (the university enrolls about 3,500 new students per year). CS 8 was chosen due to its students having little background in engineering/computing subjects, and due to our access to the course labs. The trials were the first lab assignment for CS 8, held in the first week of the 2011 spring quarter, conducted during a scheduled three hour lab section. The trials were conducted in two 3-hour lab sections with a total of 51participants.

The participants were blind to the condition and purpose of the handout. The university's Institution Review Board approved making participation a course assignment due to the similarity between the experiment and normal assignments. Therefore, the participants were not volunteers, but instead were required to participate as part of their coursework, thus reducing self-selection bias. The participants were not given grade information related to their performance on the experiment.

The participants initially completed a survey with questions related to technical competence. Each question was on a scale of 1 to 5, where 1 was defined as "little/none" and 5 as "a lot." The prompt, "To what extent have you used programming languages, such as QBASIC, C/C++, Java, Python, Matlab or Mindstorms?" resulted in 68% of participants reporting 1 (little/none), 18% reporting 2, and 14% reporting 3. No participants reported 4 or 5. No significant correlation was found between usability results and self-reported technical competence.

Figure 15: A challenge example shown to participants.



5.2 Design

The experiment contained two trials. The first trial asked 25 participants to program a solution for three challenges. The second trial asked 26 participants to program a solution for two challenges. Between the trials, one question was the same. Thus, the experiment contained 51 participants and four challenges.

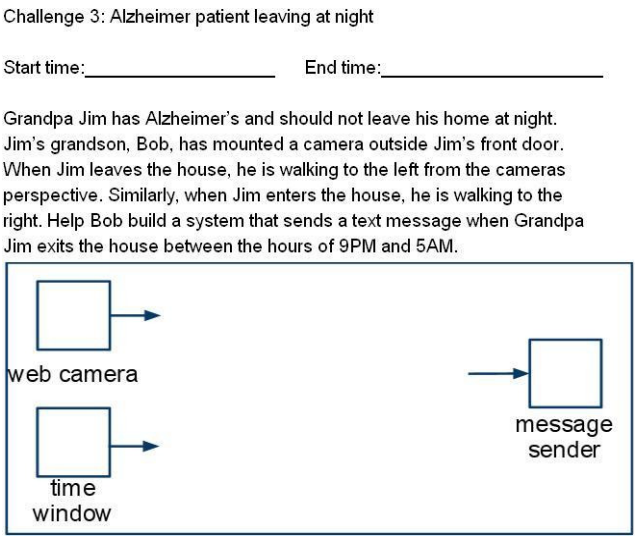
5.3 Materials

The experiment had three components: A website, a handout, and the EasyNotify environment. EasyNotify is a web-based software tool for using MNFL. The website remained available during the trials. The website had an overview picture and a link to an instructional video [8], which described the EasyNotify environment. The overview picture indicates the locations of the instantiation buttons for available audio/video blocks and for MNFL compute blocks. 15 such blocks were available: combine, compare, counter, math, n-combine, once-yes-stays-yes, opposite, pulse generator, stopwatch, threshold, yes prolonger, direction extractor, fall extractor, motion extractor and sound extractor. The one-minute instructional video has no audio, and demonstrates how to instantiate, manipulate, and connect MNFL blocks.

The handout contained instructions and challenges. The handout instructed the participant to work alone and explained how to access the EasyNotify environment. Each challenge included a challenge title, a start time field, an end time field, a problem description, and a solution field. The handout had an example challenge, shown in Figure 15, illustrating how to fill in the three fields. Following the example challenge were the challenges. The first trial had the following three challenges:

1. Homeowner Sue wants to build a fall detection system for her aging mother, Betty. Sue has mounted her web camera to look at the main living area in Betty's apartment. Help Sue build a fall detection system that sends her a text message when Betty falls.
2. Homeowner Lisa is hard of hearing and wants to build a visitor detecting light inside her home, so she knows when someone is at her door. She has mounted her web camera to look at the space in front of her front door. Help Lisa build a visitor detecting system that will turn on a light inside her house when someone is at her door.
3. (Shown in Figure 16) Grandpa Jim has Alzheimer's and should not leave his home at night. Jim's grandson, Bob, has mounted a camera outside Jim's front door. When Jim leaves the house, he is walking to the left from the cameras perspective. Similarly, when Jim enters the house, he is

Figure 16: A challenge provided to participants.



walking to the right. Help Bob build a system that sends a text message when Grandpa Jim exits the house between the hours of 9PM and 5AM.

The second trial had the following two challenges:

1. (Shown in Figure 15) Grandma Lauren has an emergency button (button 1) that, when pressed, trips a beeper that signals for help. If she accidentally trips the system, then she can reset the system with a second button (button 2). Help Grandma Lauren build an emergency beeper system.
2. Homeowner Lisa is hard of hearing and wants to build a visitor detecting light inside her home, so she knows when someone is at her door. She has mounted her web camera to look at the space in front of her front door. Help Lisa build a visitor detecting system that will turn on a light inside her house when someone is at her door.

The EasyNotify environment contained the blocks in Figure 6 and a single audio/video block. The single audio/video block was a looping video of a person walking and falling to the left, then a person walking to the left.

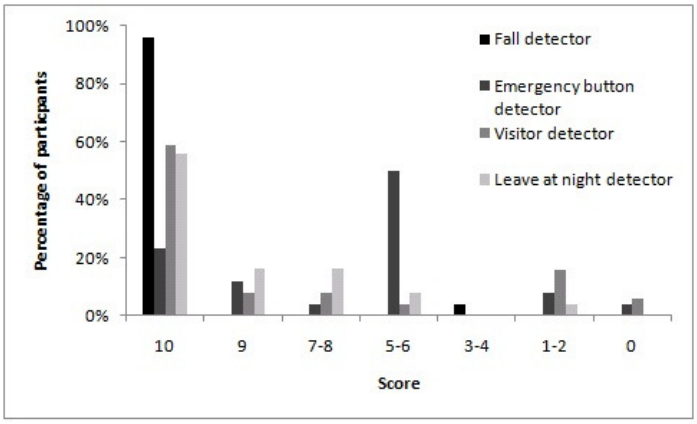
5.4 Procedure

The experiment contained the following procedures:

1) The experimenter instructed participants to review the instructional video and the overview picture at the website, and review this material as the participant saw fit during the trials.
2) The experimenter individually distributed the handout to participants, and instructed the participants to implement the example challenge in the EasyNotify environment before working on the challenges.
3) After three minutes, the experimenter described the procedure for filling in the start time field, end time field, and solution field.
4) The experimenter instructed participants to start the challenges.

The experimenter replied to questions by reminding the participant to review the instructional video and overview picture. The regular teaching assistant left the room. Handouts were graded by a hired undergraduate. *The grader was blind to the condition and purpose of the handout.* Each challenge was graded using the following scoring rubric:

Figure 17: The distribution of scores for the four challenges. The visitor detector challenge had twice as many participants.



10	Valid, working solution.
9	Almost valid. Solution works most of the time.
7-8	Used the right blocks but connected incorrectly.
5-6	Used mostly the right blocks but has extraneous/wrong blocks.
3-4	Used one of the right blocks but has mostly wrong blocks.
1-2	Seems like participant did not understand the desired result, very confused. Answer is basically nonsense.
0	Blank or completely wrong, does not use blocks, etc.

5.5 Results

Some challenges required more blocks to build a correct solution than other challenges. The order of challenges that required the least blocks to the most is: fall detector (1 block), emergency button detector (1 block), visitor detector (2 blocks), and leave at night detector (2 blocks). The average scores were 9.72, 6.42, 7.68, and 8.8, respectively. The average participant time was 4.8 minutes, 8.4 minutes, 5.5 minutes, and 12.4 minutes, respectively.

Figure 17 shows the percentage of participants that achieved each possible score. Nearly 100% correctly programmed the fall detector challenge. Only 35% correctly (10 points) or nearly correctly (9 points) programmed the emergency button detector; however, 89% at least used mostly the right blocks but had extraneous/wrong blocks. The emergency button detector was the only challenge to require a state-based solution. We suspect the state-based solution requirement was a strong factor in the low score for the emergency button detector. 76% correctly or nearly correctly programmed the visitor detector challenge. 72% correctly or nearly correctly programmed the leave at night detector. Researchers are encouraged to contact the authors to gain access to the complete collection of scanned handouts from the usability trials.

A valid solution for detecting a person leaving home at night, as described in Figure 16, is shown in Figure 18. The web camera block is connected to a direction extractor block. The "Left" output of the direction extractor block and the time window block are connected to a combine block using AND logic. The combine block is connected to a message sender. Therefore, if there is motion to the left (indicating someone is leaving) and the current time is nighttime, then a message is sent.

An invalid solution is shown in Figure 19. The web camera block is connected to a direction extractor block; however, the

Figure 18: Valid solution designed by participant for the same problem as Figure 16, person leaving home at night.

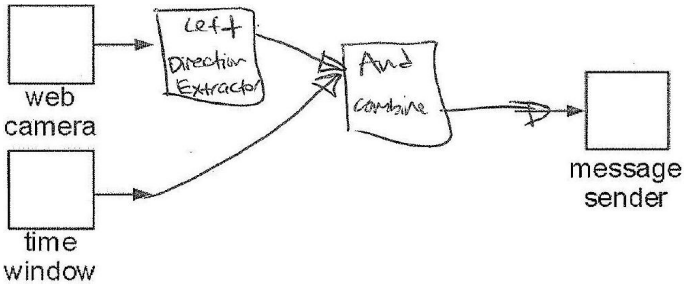
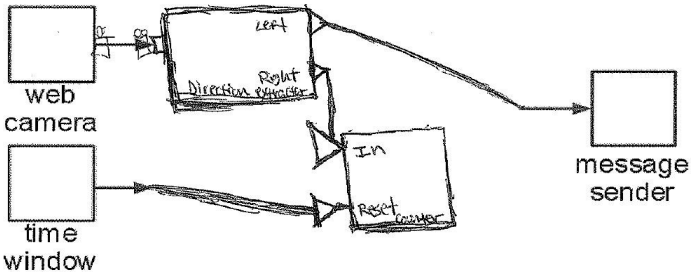


Figure 19: Invalid solution designed by participant for detecting person leaving home at night.



direction extractor's "Left" is directly connected to the message sender. Therefore, a message is sent when someone leaves, but not necessarily only at night. The participant connected the direction extractor's "Right" output to a counter block's "In" input, which increments on a rising edge. The time window block is connected to the counter block's reset. With the counter block connected in such a way, the resulting logic is similar to a tripper, which is not the correct behavior all the time.

6. Limitations and directions

Advanced time-oriented behavior is hard to capture in MNFL. For example, excessive wandering, in which a person continually walks between rooms for an extended period of time (commonly indicating disorientation), requires the end-user to determine the time a person has been in each room and use that information in two ways: reset the excessive wandering timer when the person has settled, and maintain the timer otherwise. The complexity increases with the number of rooms. Other programming approaches, such as state machines or sequential instructions, may be more appropriate for such behavior.

Further exploration of usability is needed to determine a learning curve and a limit of usability by end-users. Although the current results are promising in that common use-cases are programmable by end-users, there is a ceiling effect on the data that is preventing a clear indication of the limits of MNFL's usability. Also, the poorer participant performance on the emergency button detector challenge needs further investigation.

The built-in blocks are designed to cover most in-home monitoring applications; however, we do not claim to know all blocks that end-users will need. We are working on making programmable blocks of three sorts: hierarchical blocks, state-machine-based block behavior programming, and C/C++ based block behavior programming. A hierarchical block would simply contain pre-defined blocks. The state-machine and C/C++ based block behavior programming would allow the end-user to write a custom block.

7. Conclusions

We introduced the Monitoring and Notification Flow Language (MNFL) for capturing assistive monitoring applications. The examples showed how common assistive monitoring applications could be captured via MNFL, and how customization of those applications to unique or changing requirements could be made. Initial usability trials demonstrate the immediate intuitiveness of MNFL—with nearly no training, lay users could more than 50% of the time successfully select the necessary 1-2 blocks needed to solve a problem, spending less than 8 minutes on average to read the problem, find/instantiate/connect the right blocks, and test the system. Future usability trails, likely including a short training tutorial, template systems, online hints, or some combination thereof, is needed to further understand the potential and limitations of end-user assistive monitoring programming with MNFL.

References

[1] Berry, G. G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. Science of computer programming (1992). Volume 19. Issue 2. Pg. 87.

[2] Boussinot, F. R. de Simone. The Esterel Language. Proceedings of the IEEE (1991). Volume 79. Issue 9. Pg. 1293.

[3] Carmien, S., End User Programming and Context Responsiveness in Handheld Prompting Systems for Persons with Cognitive Disabilities and Caregivers. In Extended Abstracts of CHI 2005, (Portland, Oregon, 2005), 1252-1255.

[4] Cotterell, S., F. Vahid, W. Najjar and H. Hsieh. First Results with eBlocks: Embedded Systems Building Blocks. ACM/IEEE ISSS/CODES conference, 2003, pp. 168-175.

[5] Cotterell, S., R. Mannion, F. Vahid, and H. Hsieh. eBlocks - An Enabling Technology for Basic Sensor Based Systems. In Proceedings 2005 Fourth International Symposium on Information Processing in Sensor Networks, IEEE, 2005, pp. 422-427.

[6] Dey, A.K., R. Hamid, C. Beckmann, I. Li, and D. Hsu. a CAPpella: programming by demonstration of context-aware applications. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (CHI '04). ACM, New York, NY, USA, 33-40.

[7] Dey, A.K. Understanding and Using Context. *Personal Ubiquitous Comput.* 5, 1 (January 2001), pp. 4-7.

[8] EasyNotify Challenge instruction video. http://www.youtube.com/watch?v=ZUZDBX9J5Bk. April 2011.

[9] Edgcomb, A., and F. Vahid. Monitoring and Notification Flow Language. May 2011. http://static.cs.ucr.edu/store/techreports/UCR-CS-2011-051000.pdf.

[10] Eizmendi, G., and G.M. Craddock. Challenges for Assistive Technology. Association for the Advancement of Assistive Technology in Europe conference (AAATE), 2007.

[11] GrandCare. http://www.grandcare.com/. June 2011.

[12] Harris Communications. http://harriscomm.com/. June 2011.

[13] Jahnke, S.H., M. D'Entremont and J. Stier. Facilitating the programming of the smart home. Wireless Communications, 2002.

[14] Kahn, G. The Semantics of a Simple Language for Parallel Programming. Information Processing 1974. North-Holland Publishing Company, 1974.

[15] Lego Mindstorms. http://mindstorms.lego.com/en-us/ default.aspx. June 2011.

[16] Lighthouse International. http://lighthouse.org/. June 2011.

[17] MaxiAids. http://www.maxiaids.com/. June 2011.

[18] Microsoft Visual Programming Language. http://msdn.microsoft.com/en-us/robotics/default.aspx. Apr. 2011.

[19] Motorola Homesite. http://www.smarthomeusa.com/ Products/Homesight-USB-Kit/manuals/homesight-software-user-guide.pdf. June 2011.

[20] Philips, B. and H. Zhao. Predictors of Assistive Technology Abandonment. Assistive Technology, Vol. 5.1, 1993, pp. 36-45.
[21] QuietCare. http://www.careinnovations.com/Products/ QuietCare/. June 2011.
[22] Rashidi, P. and D.J. Cook. 2009. Keeping the resident in the loop: adapting the smart home to the user. *Trans. Sys. Man Cyber. Part A* 39, 5 (September 2009), 949-959.
[23] Resnick, M., J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman and Y. Kafai. Scratch: programming for all. Comm. of the ACM Volume 52, Issue 11. November 2009.
[24] Resnick, M., S. Ocko, and S. Papert, LEGO, Logo, and Design, Children's Environments Quarterly 5, No. 4, pg. 14-18, 1988.
[25] Riemer-Reiss, M. Assistive Technology Discontinuance. Technology and Persons with Disabilities Conference, 2000.
[26] Scherer, M.J. The change in emphasis from people to person: Introduction to the special issue on Assistive Technology. Disability and Rehabilitation, 2002, vol 24.
[27] SmartHome. http://www.smarthome.com/. June 2011.
[28] Sousa, J.P., B. Schmeri, V. Poladian and A. Brodsky. uDesign: End-User Design Applied to Monitoring and Control Applications for Smart Spaces. Working IFIP/IEEE Conf on Software Architecture, 2008.
[29] Truong, K., E. Huang, G. Abowd. CAMP: A Magnetic Poetry Interface for End-User Programming of Capture Applications for the Home. UbiComp 2004. pp. 143 – 160.
[30] ViaSee. http://viasee.com/. June 2011.
[31] Wallich, P. Mindstorms Not Just a Kid's Toy. IEEE Spectrum, Volume 38, Number 9, September 2001.
[32] WellAware. http://www.wellawaresystems.com/. June 2011.