Control Software for Home Automation, Design Aspects and Position Paper

D. Greaves

Abstract

Keywords: Autohan, UPnP, Iota, XML.

The home is an eternal, heterogeneous, distributed computing environment which must be secure and reliable. Computers and embedded processors in the home are all different shapes and sizes and ages. Hence the home poses one of the most challenging environments for co-operative programming. We envisage that control software is introduced into the home by four different methods varying from embedded ROM code to applets generated from a combination of natural language and gesture with wands. But we argue that, in the long term, all of it must be represented in a common, formally-verifiable language and conform to a common scripting convention. The AutoHan project at the University of Cambridge Computer Laboratory is the umbrella under which we are trying to grow these ideas [3].

1 Introduction: Application Code versus Data Processes

The number of microprocessors in the western home of today can easily exceed one hundred. In the past, each processor runs ROM code that encapsulated a fixed application that was never upgraded. An application was typically a simple event control loop, which for instance, ejected the CD player draw when the eject button was pressed, and implemented a rule of not much greater complexity for each of the other buttons on the device or on the device's dedicated IR controller. However, as devices become networked the interaction of these 'canned' applications becomes important. In addition, the presence of server platforms in the home is growing. The modern VCR in the form a TiVo [1] is in fact a Linux PC capable of hosting any application. For the purpose of this paper, we will redefine the term '*application*' to mean a control program that does not do any data processing. Hence a program that, for example, implements a specific functions on stream of data, such as transcoding from MPEG to M-JPEG, is not an application, it is a merely a process that can be run on any suitable host with available network and processor bandwidth. We assume that the user interface to this transcoder, that sets up its parameters and sets which streams it operates on, is replaced with a true application, in our sense, that communicates using UPnP SOAP and GENA [13] with the transcoder process. A large library of data processors will be readily available for running in the home environment of the future. Many will be invoked with application control software and others may run eternally. Functions provided by members of the process library include speech recognition, pay-per-view credential checking and electronic money handling. Some processes are essential to the correct operation of the home control software. These include interpreters for the language(s) that the control software is scripted in and a secure database of information about the home.

Therefore we are arguing that all embedded applications will export the details of their operation, or their 'semantics' over the network. This is a natural extension to what is offered in UPnP for exporting device description and providing device control. UPnP enables a device to describe itself and offer an API to allow remote applications to invoke its internal functions. However UPnP does not export the canned application inside a device and so there is no knowledge of what a device will actually do when introduced to the environment. Its internal, canned application will typically remain running, hosted on the internal processor in the device. Only if a device exports information about the control code that it introduces into the home can the home remain stable.

Below we present our view that that there will be four main sources of home control software before looking at design issues in detail. We are working on the basis that all of the control software will be written in common language that is friendly with XML and amenable for formal proof.

2 Sources of Application Code

We think that four sources of application code will be used in the home of the future:

1. *Standalone, built-in ROM applications.* Most conventional consumer devices are expected to operate out-of-the-box without connection to any home network.

The canned application in the internal ROM provides the connection between the user-interface portion of a device and its internal API. As standard, both of these halves are assumed to be described in XML and controllable using SOAP according to the UPnP specification.

2. *Networked, built-in ROM applications.* This second form again operates out of ROM provided in the devices but is different in that it detects that a peer device or service is accessible over the home network connection and so starts to use it or offer a service based on it. A simple example is that a pair of hi-fi amplifiers might detect each other and allow routing of audio from one to the other. The user interface to these built-in applications is also built in, and in the amplifier example, could take the form of an additional item on the source selector menu or rotary switch, which is 'remote amplifier'. The user simply selects this source to have access to the sound channels selected with the source selector on the other amplifier. Similar example applications using video and the relaying of commands from infra-red remote control handsets are also likely to be built-in to ROM.

3. *Uploaded and Download Applications.* These are software programs or modules which move over the network to their point of execution. There are three main forms: 1) modules which are downloaded into a device and then provide additional functionality for that device, 2) modules which are stored in the ROM of a device and are exported to be executed on a more powerful platform, and 3) modules which are downloaded from the WWW or by a remote service technician over the access network.

4. *User Applications.*

   User applications have been created by the user. Although they are not different in architecture from downloaded or uploaded applications, the creation of them is via novel user interfaces, based on gesture, natural language and a new interface paradigm. They could also be created by AI techniques that monitor the behaviour of the user and suggest new automations. We have been working on two scripting forms. Firstly we have been working with SRI as part of the project 'Dialogues in the Home Machine Environment' to generate scripts to control the home and secondly we have created programs from the 'Media Cubes' described below.

3 A suitable language ?

The crux of the issue is to chose a suitable scripting language. Java is an imperative language with a reflection API and is worthy of consideration. However, we have previously written about the shortcomings of strongly-typed synchronous RPC as the basis for eternal and flexible systems [2]. Instead we believe that functional and declarative languages are easier to reason about and that a higher-level, more domain-specific language will be more practical. We have tried two approaches so far: CEL and Iota. Both of these have been tied to XML since XML is the universal language for extensible data representation. Extensible data formats are essential for eternal systems such as home control. It is impossible for systems that use binary-coded data representation, such as HAVi [12] to interoperate without reference to a shared header file or set of RPC stubs. Such a shared reference will soon go out of date in that it will not include the codepoints for newly invented functions and devices.

The utility of readily extensible systems comes from that, as time passes, knowledge is disseminated about how others have extended the system and this eventually becomes set in stone. In the same way that we are all used to the rather arbitrary string 'C:' denoting the root drive on a PC (which may be a partition and not a separate disk or indeed it may be an entirely virtual PC), home control software will gradually evolve to rely on certain structures to always be present in the XML that describes a home, and to rely on strings such as 'FrontDoorBell', 'MediaServer' or 'LivingRoom'.

The major design considerations for our application language are now listed:

1. *Basics.*

   Obviously we need a language that supports concurrency and network communication and secure loadable code modules. It should either have a portable bytecode or be efficiently interpretable in source code form.

2. *Efficient handling of asynchronous, parameterised events.*

   Many home automation applications will operate in response to events generated by other applications or devices that asynchronously disseminate the events. An example is to switch the hall lights on if the front door is opened and it is dark. Asynchronous dissemination is used since this is efficient for nearly-reliable multicast and the generator does not wish to block if a previously interested application has suddenly gone away.

3. *Type-safe handling of XML.*

As has now widely accepted in home automation projects, AutoHan keeps all of the information about the home in a secure, distributed, federated or centralised XML database with secure access. This is called DHan and is described in [3].

One raison d'etre for XML is that it is a convenient format for tree-structured data that can be viewed and understood, at least to some extent, by a viewer, person or application that does not have full knowledge of the formal structure of the tree. Instead, the recipient may have partial knowledge, or knowledge of a previous release of the structure definition, or may infer the structure and meaning from the direct use of English and ASCII in the XML format. Parts of an XML tree that are not recognised can be ignored while still correctly parsing other parts of the same XML document. An example is that XML can be viewed in a web browser with or without style-sheets to define the view.

Our programming language requires tight integration with the home XML database in that most of the data values and identifiers in the language will originate in the database. XML itself is purely an annotated tree, but schemas can be defined to impose further structure on required fields, values and referential consistency [4]. We are believers in strongly-typed programming languages, but equally we wish to allow the format of the XML data controlled by the program to be flexible. This is a dichotomy. Any language design must chose how much to tolerate by default and how much to leave the 'user' to provide explicit defaults or handle as exceptions. We envisage a standard library that provides all default handlers needed for a newly mechanised home.

Another aspect of eternal systems is to provide persistent data structures that retain values over process migration, reboot and version upgrades. We think it is desirable to implement so-called 'orthogonal persistency' where no special user programming is required to achieve this. Clearly, the XML database provides a natural place for this state to be mirrored and so is another aspect of XML integration.

4. *Canned code can re-hydrate itself.*

   Re-hydration, or binding, may be considered as a process of mapping textual identifiers found in the source code to actual objects found in the execution environment, and taking default actions if there is no mapping to hand. Additional code sections may need to be loaded from the Internet or elsewhere as a side-effect. Each code section must be securely signed and authenticated. All programming languages today must implement this, but for the home, where these mappings are more dynamic and varied from one house to another, the language must make this very easy. The approach of popping up a web page that presents a kind of XML plugboard to allow the user to manually resolve and bind unknown identifiers seems good. It can leverage the ASCII nature of XML but relies on a suitable user interface already being in operation.

5. *Code may be replaced or upgraded while executing.*

   When installing new code, some of the functions of the new code will be replacements or overrides for existing functions. It may be a complete overlap if an application is being upgraded to a new version or it may be a partial overlap if a new application, such as 'follow-me audio and video phone' overrides a previous application, such as 'play audio on speakers in current room'. State from the previous application instance normally needs to be preserved. Identifying the superseded components is hard to automate and can only be done if the system as a whole has detailed knowledge of the functions of each application. It seems inevitable that any solution will require at least some attention to this problem by the application creator but the programming language can perhaps go a long way to help.

6. *Programs in the language can be created by tangible interfaces.*

   When speech, gesture and AI are used to create new scripts, it is likely that a set of phrases or macros in our language will essentially be stitched together. However, it may be sensible for the language itself to support most of these macro functions (near-)natively. This leads us to consider a highly-domain-specific language where important home concepts such as rooms, times, temperatures, people and television channels are all native data types.

7. *The behaviour of a program can be readily understood by automated reasoning.*

   We envisage that statutory regulation and contractual agreements will cover many aspects of the operation of the home network. We envisage 'serious' uses of the equipment in the home, such as the burglar alarm presented in example below, are envisaged alongside frivolous applications, (such as a program perhaps found on a freebie smartcard in a cornflakes box that turns the house lighting into a disco sound-to-light system.) Therefore it is essential that priorities between competing applications can be determined and potential conflicts determined before they arise. In addition, we will wish to impose a large set of rules on the system regarding the allowable reachable states and perhaps the rate of expenditure of children's electronic pocket money. Apart from not disabling the burglar alarm, a base rule set may restrict the user from locking himself out of the house or spending more than 100 pounds per day on video network services.

3.1 Examples languages to date

We have written various home automation apps in C and Java, but not with a view to achieving the above goals. We have also written automation apps in CEL and Iota.

Cambridge Event Language (CEL) [5] is a set of algebraic operators that match sequences of parameterised events. Our event engine stores and concurrently executes a number of event expressions composed of these operators. Events can be received over the network in GENA form or generated from the matching (firing) of event expressions themselves. Other actions can be associated with a firing of an event using an event script. Hence an application is a list of event expression and action pairs. An action can be to send a SOAP RPC to a device or send an HTML NOTIFY to the XML database to change or refresh the state stored there. Asynchronous event handling and concurrency are innate in this language. Rehydration can be implemented using a set of rules that macro-generate the actual event script from a canned form by expanding it against information currently available in the XML database and by repeating this process as needed when new information is registered in the database or certain changes occur.

Formal proof regarding reachability and interference between applications is being done for CEL by M Rowbotham at Cambridge. He has taken a number of canned applications for home devices and added some hand-written applications that link the home devices together to perform an integrated function. All code fragments are written in CEL. A program performs static analysis of the whole body of CEL and generates verification conditions. The conditions are output to a n-SAT solver where n is the level of quantization of the most fine variable in the system (e.g. we might support only 16 different temperatures for a room to keep n small). The important verification conditions all seem to relate to the rate of possible generation of events. For instance, a simple rule that prevents event storms can be expressed as a time limit restriction on how soon the result of a given event will again cause that event to occur. Equally, a rule that restricts expenditure of pocket money can be expressed as a rate restriction on debit transactions. Finally, reachability can be expressed as an infinite time before some event occurs, such as a desire to prevent a given condition ever happening. This work is due to finish in time for the conference. An extension will be to generate HOL [6] theorems for proof rather than restrict to SAT cases.

3.2 Example Language: Iota.

Bierman and Sewell invented Iota as an experimental language for manipulating XML [7]. The language is functional and performs type-inference in the same way that SML does [8]. XML code is directly embedded as patterns or expressions in Iota. For example the following is a valid expression in Iota

```
<meaning of="life">7*6</meaning>
```

The integers could be replaced with variables whose values would be fetched, but as written the constant 7 is multipled by the 6 to produce a fragment that cannot be further evaluated: a constant expression.

Iota provides the powerful facility of pattern matching. This enables compact and highly readable code. For example the following piece of code accepts a piece of person markup in XML as an argument and returns its age attribute value.

```
fun returnAge <person age=a> => a;
```

Concurrency in Iota is either provided outside the language when an external agent starts more than one program running at once, or else internally using the built-in features of Pi Calculus [9] including a parallel composition operator and synchronous message passing over named channels. Re-hydration of canned code can be implemented via subroutine calls to Iota stubs that indirect through the main XML database.

Because Iota has a formal specification based on SML, Pi Calculus and other mainstream ideas within our theory group, there is already a lot of experience to hand for proving properties of Iota programs, but I have yet to see any proof results when Iota is used for home applications.

Our experience with generating applications from the tangible interfaces suggests that rules are a good basis for representation (e.g. when I press THIS button turn on THAT object), and hence there was an obvious mapping into CEL. With Iota, response to asynchronous events is merely a matter of forking a subprocess that blocks on each possible source of an asynchronous event. Iota provides very lightweight concurrency for this purpose and it is an interesting question of theory meeting practice to see whether the synchronous concurrency is indeed sufficiently lightweight for an implementation to scale in the same way as has been achieved with CEL.

As already mentioned, when it comes to designing a programming language to handle XML, the weak-typing benefits of XML must be respected, otherwise the language has achieved nothing. There are a number of levels of embedding XML into a programming language, and Iota has currently taken a level where the structure of the tree is not checked for any formal structure, apart from via matches failing at runtime. Iota also considers strings to be atomic, i.e., strings that are not the same, but textually similar are considered totally different. Also, the order of sub-trees in a list is considered important by default, whereas there are many applications of XML where the order will need to be ignored. Further experience will show whether Iota has taken the correct approaches. We must recognise that methods of generating and checking Iota programs may be able to provide additional richness, rather than building features into the language. Iota is radically different from XMILE, an imperative language whose concrete syntax is in XML [14].

4 Operating Systems and UbiqtOS

We are envisioning a system that is homogeneous at the application scripting layer, although clearly there are a large number of physical networks that can carry the required UPnP and AutoHan protocols as well as the media and other application data. Clearly execution platforms are different in capabilities, but by our restriction of the definition of an 'application', mentioned earlier, it is fair to say that nearly any platform will be able to execute nearly any application. We envision the processing power needed for the interpreter for a small number of applications being below one MIP.

Saif has generated an operating system called UbiqtOS [10]. This has taken a slightly different approach from that outlined above and makes interesting comparison. Saif assumes an XML database of all relevant information and similarly allows this to be federated between the different execution platforms that make up the system. All application and protocol code apart from the kernel itself is written in Java bytecode and each IPC and network operation between concurrent processes is indirected by the kernel through the XML database where information about how to implement the operation in detail may be stored. For instance, depending on the database contents, a GENA notify may be sent over IPV4, IPV6, UDP, TCP or as a raw network datagram. The application code that generated the GENA event would not know which. The kernel provides full support for weak mobility of active agents (i.e. suspended Java programs sent over the network) and these agents can update the XML database to configure the system.

Saif's view of an eternal ubiquitous system places a small kernel and JVM in ROM in each device. He assumes this will never need to be changed but allows that all other aspects of the system can be upgraded by indirections stored in XML. This provides one of the most flexible environments for embedded software ever created. Within AutoHan, such a platform has a clear role for the data processors described in the introduction. It can also be used as the underlying operating system for the application script interpreter where, no doubt, wire protocols and requirements will change from time to time. However, for the application interpreter itself, we do not envisage much evolution, and so do not wish to call on the available flexibility at the top layer. Instead, we want our interpreter to assume a similar status to the JVM, or indeed, we could compile our script to Java bytecode without loss of the clean semantics.

5 Media Cubes and Rabbit Wand

Our fourth source of scripts is user applications, so now we summarise our work with handheld controllers and voice input to generate scripts.

Infra-red is an important medium for the home and certainly people will continue to use remote controllers of increasing sophistication. Some controllers will stay put in a room and others will be carried around and worn. Some controllers will continue to be provided with consumer goods but others will be much more general purpose. We wish to use this new generation of controllers both for real-time control of the network and for programming the home to set its future actions and behaviour. A major semantic issue with the controllers of today is that they must be pointed at the object under control. In this work, we want to *point* at things not immediately to hand, such as the heating boiler, and also more abstract things, such as a radio programme on a given channel at a future date. We also want to combine controllers together or point them at each other to achieve advanced operations in a natural way.

The Rabbit Wand and the Media Cubes [11] are examples of the new controller generation. The Media Cubes provide something to point at to iconify the abstract entities and then go further, when we combine cubes to form a programming language. The Rabbit Wand is an IR controller with a microphone input and both narrow and wide infra-red beams. When the narrow beam is a visible laser pointer, or similar, very accurate pointing is possible. We might envisage a consumer device with a number of little targets that can be pointed at when it is occasionally necessary to do some fine control function.

The controller can be used to point at devices and the speech input function processes demonstrative adjectives for correlation with the devices being pointed at. A motion sensor or solid-state gyro could be included cheaply in the rabbit.

The first Rabbit does not do speech recognition itself, but instead puts the voice over a DECT-style link to a network receiver node. The receiver node relays the audio over the home network to a server running the speech recognition function and other software. We are working on this other software.

Figure 3 shows prototype *Media Cubes* [11] that may be placed next to each other in various configurations to write

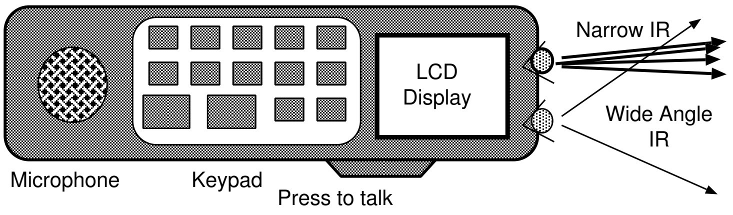
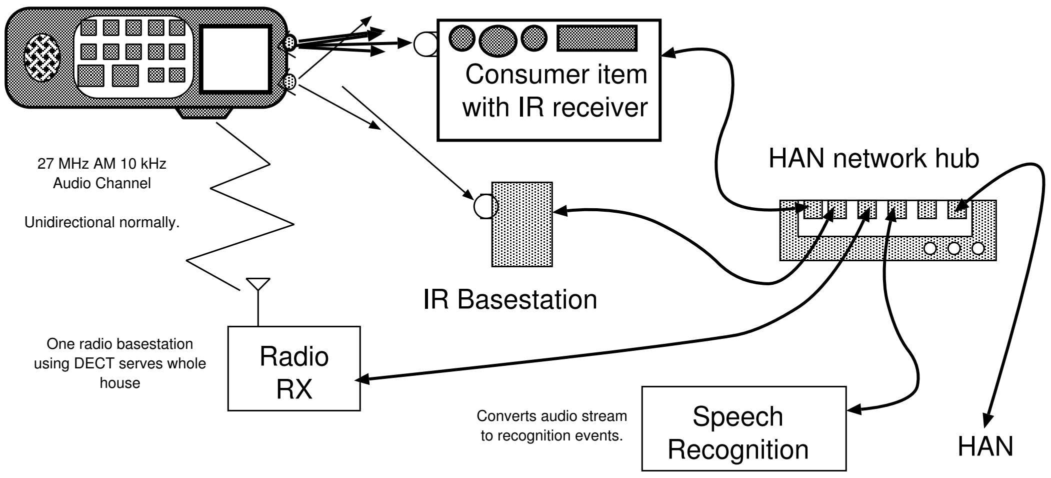


Figure 1. Rabbit IR controller with Speech Input.



Consumer item with IR receiver
HAN network hub
27 MHz AM 10 kHz Audio Channel
Unidirectional normally.
IR Basestation
One radio basestation using DECT serves whole house
Radio RX
Converts audio stream to recognition events.
Speech Recognition
HAN

Figure 2. Rabbit network architecture.

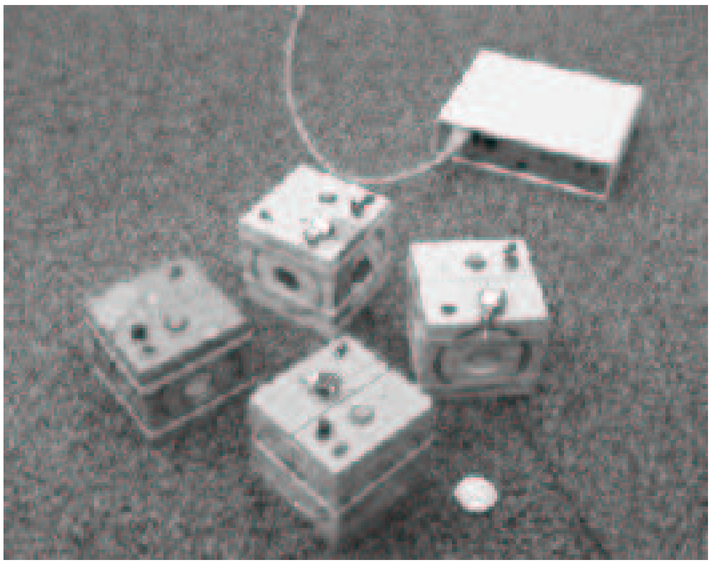


Figure 3. Prototype Media Cubes.

programs and implement real-time home control. They can also be pointed at by an IR wand. These first media cubes are indeed cubic and have one button. This is deliberately very different from an IR controller. We expect our final controllers to be somewhere in between the two, preserving the power and tactile nature of current IR controllers while supporting the adjacency facilities of cubes.

The Media Cubes pictured here are early prototypes. Each cube has a bidirectional infra-red link to the IR basestation (also pictured), and can be pointed at with the Rabbit Wand or other IR controller. The face of each cube has an induction coil, which allows it to detect adjacency with other cubes. These coils should be covered over with graphic sheets that indicate the function of each face of each cube, but these sheets are missing in the photo.

Having induction coils, batteries and IR transceivers on each cube is not seen as a long-term technology solution. These prototypes are just to experiment with the cube language and concept. In the long run, cubes are intended to be much cheaper than today's infra-red controllers. We also envisage virtual cubes which can be manipulated on a display device and that home programming information can be displayed with cubes.

Here is an example use of the cubes, developed by R Hauge. If we point the television IR controller at the TV and press '1' it turns on the TV on BBC-1. If instead we point the same controller at the 'time' cube and set the little display on the time cube to some time tomorrow, then pressing the '1' button on the IR controller defines a one-time program to turn on the TV on BBC-1 at that time tomorrow.

If we point the controller at the 'when' cube and place another cube which has been associated with receiving DTMF messages by telephone on the correct face of the 'when' cube, then we can generate a program to turn on the TV on channel 1 when we dial in with a given code.

The cubes may be thought of as infra-red remote controllers with one or two buttons. They are at the extreme other end of the spectrum from our current infra-red controllers that bristle with buttons. We shall soon have made some more combi-cubes that have more controls on them yet which can still be combined with each other.

6 A Scripting Example

Figure 4 shows the basic elements on a network burglar alarm and hi-fi system that interact owing to sharing the same audio modules. The part of the overall system we shall consider is how to mute the hi-fi without over-riding the burglar alarm. The muter and the burglar alarm are separate sections of application code, written at different times by different authors and loaded into the system at different times. Our system ensures that they co-operate correctly, in that we have a rule of consistency that states that the burglar-alarm subsystem must be present and in working order at all times.

We shall not consider the source of the mute, which might in practice be the front-door bell or an incoming phone call. We just consider an event source that generates a 'mute' event from time to time. Similarly we have an alarm senor that generates a 'burglar' event from time to time. We do not consider um-muting or turning off the burglar alarm.

The Alarm Tone Generator is potentially a software object running on any platform in the home. It accepts 'start' and 'stop' events and requires its output to be connected via a multi-media stream to the audio output module. A similar stream is needed to connect the CD player to the audio output module. These streams are created and destroyed by sending events to the connector. The Alarm Tone Generator needs to be instantiated when the burglar alarm application is installed or when it goes off. The other objects are assumed to exist already and to be registered in the registry already.

Each application is essentially a one-liner in terms of event propagation:

Burglar Alarm When 'burglar' is detected, send 'start' to Alarm Tone Generator.

Muter When 'mute' is detected, set volume on Audio Module to zero.

The critical issues is whether the Burglar Alarm also does the following

Burglar Alarm When 'burglar' is detected, also set volume on Audio Module to maximum.

if it does not, then the muter application should not be allowed to start, since it would potentially defeat the burglar alarm. Apart from the volume setting on the Audio Module, similar considerations apply to the creation of the multi-media links.

The multi-media links may be physical wires or channel connections over an ATM or Firewire network. They may also be connectionless flows over a simple Ethernet, in which case they have no actual manifestation in the network. They may be flows over a concatenation of different network types. The 'Connector' is a system function that enables events to control multi-media flows. To establish a connection in our system, a connect event is sent to the connector.

A particularly tricky part of our system design is to ensure that newer models of, say, the Audio Output Module do not defeat the burglar alarm. Our aim in using XML to describe these devices is that tags that are exported by a device but not mentioned in an application script can be safely ignored. However, if the newer version of the module has a new tag that essentially acts as a hidden mute function, then our system can break. Therefore, rules for extending devices need to be respected by the maker of such new devices.

We enlarge the burglar alarm definition beyond what is shown in the picture by requiring it to capture 'burglar' events from any Burglar Sensors stored in the registry, not just one. Similarly, we want to sound the alarm on all devices that can create the required audio outputs (such as headphones the user is wearing). Finally, modules such as the Tone Generator and the Audio Output Module, in general, may or may not have the capability to multiplex their multi-media streams from or to multiple peers, and so we wish to support a layer of abstraction or other means that automatically introduces this capability where provide in the physical devices (such as in our Warren-based implementation of the multi-media parts).

Figure 5 shows the code for the Burglar Alarm application. This code is fired up at system boot time by a statutory section of the boot sequence. We assume reboot is less than once per year. The sensors and Audio Modules have, we envisage, a shorter lifetime than the Burglar Alarm application: that is, they can come and go between reboots and hence between restarting the Burglar Alarm script.

Our aim is to fully complete the implementation of this system so that a correct way of defining device and application properties evolves.

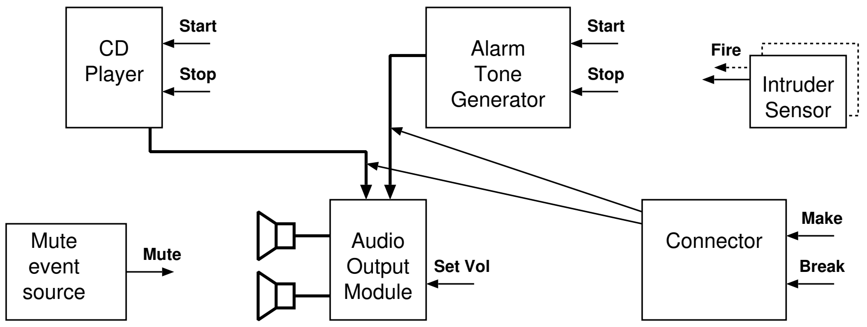


Figure 4. Burglar Alarm and hi-fi Example.

process: for all Burglar Sensors, if there is a burglar execute procedure p.

procedure p: for all audio output modules, set their volume to max, instantiate a burglar Alarm Tone generator, connect the module to the tone generator and start the tone generator. Set the module volume to max.

Figure 5. Burglar Alarm Script

process: if the muter mute event happens, set the Audio Output Volume level to zero.

Figure 6. Part of the potentially offending Muter Script

Acknowledgments

Acknowledgments are due to the following people for their contribution to the work described above. Peter Sewell, Umar Saif, Gavin Bierman, Daniel Gordon, Rob Hague, Mike Rowbotham and Alan Blackwell.

References

[1] 'TiVo automatic digital recordings of your favourite TV shows without hassles' TiVo Corporation. www.tivo.com.

[2] 'Communication Primitives for Ubiquitous Systems or RPC Considered Harmful'. Umar Saif, David J. Greaves. Proceedings of 21st International Conference of Distributed Computing Systems (Workshop on Smart Appliances and Wearable Computing), 2001.

[3] AutoHan Project Web Pages. www.cl.cam.ac.uk/Research/SRG/netos/han/AutoHAN

[4] 'XML-Schema Part 0 Primer.' www.w3.org/XML.

[5] 'Using Events for the Scalable Federation of Heterogeneous Components'. Proceedings of ACM SIGOPS European Workshop, September 1998. Bates, J. Bacon, K. Moody and M. Spiteri.

[6] HOL 'Introduction to HOL' MJC Gordon, TF Melham (eds.) Cambridge University Press 1993 ISBN 0-521-441897.

[7] 'The Iota Programming Language. What is it?' G Bierman, P Sewell. www.cl.cam.ac.uk/ gmb/Iota/

[8] 'ML for the Working Programmer' LC Paulson. , 2nd Edition. Cambridge University Press.

[9] 'The Polyadic pi-Calculus: A Tutorial' R Milner. www.lfcs.informatics.ed.ac.uk/reports/91/ECS-LFCS-91-180

[10] 'Context-aware Adaptation in UbiqtOS: A Java-based Embedded Operating System for Ubiquitous Computing'. U Saif, D Greaves Proceedings ACM SOSP 2001. Banff, Canada. See also Saif's PhD dissertation from Cambridge.

[11] 'AutoHAN: An Architecture for Programming the Home' A Blackwell, R Hague. End-User Programming Symposium at HCC 2001.

[12] 'The HAVi Specification V1.0 beta' www.havi.org

[13] 'Universal Plug and Play' www.upnp.org

[14] XMILE http://pizza.cs.ucl.ac.uk/xmile