*Kenneth J. Turner. Device Services for The Home. In Khalil Drira, Ahmed Hadj Kacem and Mohamed Jmaiel, editors, Proc. 10th Int. Conf. on New Technologies for Distributed Systems, pages 41-48, IEEE Computer Society, Los Alamitos, California, USA, June 2010, ISBN 978-1-4244-7066-2*

# Device Services for The Home

Kenneth J. Turner
Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA, UK
Email: kjt@cs.stir.ac.uk

***Abstract*—An approach is presented for flexible support of devices in smart homes, meeting the needs of both home automation and telecare. Device services are introduced as a generalisation of sensor fusion, allowing ready customisation of how sensor inputs and actuator outputs are mapped to each other. Low-level management of devices is supported, but integrated with the high-level use of goals and policies. A range of typical device services is used to illustrate the approach.**

***Keywords*—Device Management, Home Automation, Policy-Based Management, Sensor Fusion, Telecare.**

## I. Introduction

The goal of this work is to flexibly support devices in the home, with applications to home automation and telecare.

### A. Motivation

Home automation aims to let the user control a variety of devices around the home. Telecare aims to provide automated support to those receiving care at home. Both applications can be underpinned by a similar infrastructure, although the specific devices and services needed for each will vary.

The author has demonstrated that goals and policies offer a convenient way of managing the home [15], [17]. ACCENT (Advanced Component Control Enhancing Network Technologies, www.cs.stir.ac.uk/accent) is an approach and toolset for managing systems through goals and policies. The approach is intentionally high-level and user-oriented. However, flexible support is also needed at a lower, device-oriented level.

A common approach is sensor fusion, wherein the information from multiple sensors is combined to achieve a more accurate understanding of what is happening. This is useful but limited. The aim of the work in this paper has been to significantly generalise this idea to achieve several objectives:

- It is desirable to offer actuator fusion as well as sensor fusion. In general, it should be possible to define a flexible mapping among sensor inputs and actuator outputs.
- It should be easy to define and alter the logic of such a mapping. Many home systems require specialised knowledge and reprogramming to achieve an effect like this.
- Device control is required both within the home and (securely) from a remote location.
- Flexible device support should integrate well with goals and policies.

Collectively these capabilities are referred to as device services. The approach takes a service-oriented view of devices and interlinks their capabilities.

### B. Background

*1) Home Automation:* Device control underlies home automation. Many standards have evolved in this area. IR (Infrared, www.irda.org) is used to control many kinds of domestic appliances. KNX (Konnex Association, www.knx.org) derives from earlier work on EIB (European Installation Bus) that is widely used in building management. UPnP (Universal Plug and Play, www.upnp.org) is an extension of the plug-and-play concept into the world of networked devices. X10 (www.x10europe.com) is widely used to control appliances using existing mains cabling, plus support for wireless control.

The work in this paper is independent of particular devices and protocols as this level of operation is essentially a solved problem. A harder question is how to abstract from device details for a uniform interface to higher levels of control.

Many commercial packages support home automation, e.g. Cortexa (www.cortexa.com), Girder (www.promixis.com), HAI (www.homeauto.com) and HomeSeer (www.homeseer.com). These packages offer a degree of programmability, though this is usually at a low level and requires specialised knowledge of the package and the devices.

The work in this paper does not aim to compete with well-established commercial solutions. Rather it is focused on new techniques such as a flexible device framework.

*2) Telecare:* The world population is gradually ageing, with the percentage of older people (over 65) expected to rise by 2050 to 19.3% worldwide, and much higher in some countries [7]. Although the population is ageing, older people are generally healthier and more active than in previous generations. It is beneficial for older people to live independently in their own homes as long as possible. The impracticability and cost of providing sufficient care homes also makes this a necessity. This situation has been recognised by governments in all developed countries, where many national programmes are in place to promote the use of telecare technologies.

Telecare uses computer-based systems that support delivery of care to the home. They can give the user advice, identify situations that may need intervention, reassure family members and informal carers, and relieve professional carers of low-level monitoring tasks. Telecare systems should be appropriate (from different stakeholder viewpoints), customisable (for specific user needs), flexible (offering a range of solutions), and adaptive (as care needs and conditions evolve).

Unfortunately, most telecare solutions are proprietary and relatively fixed. Changing functionality often needs specialised technical knowledge and reprogramming. Telecare standards are also immature due to the relative newness of the area.

Because telecare is an emerging discipline, this paper has the potential to have more impact. In particular, its emphasis

on adaptable device control is particularly relevant.

*3) Component Frameworks:* A variety of frameworks have been developed to allow flexible combination of components. Approaches include ADLs (Architecture Description Languages [12]), OSGi (originally Open Services Gateway initiative, www.osgi.org), SCA (Service Component Architecture, www.osoa.org), SOA (Service Oriented Architecture [10]) and SODA (Service-Oriented Device Architecture [6]).

BPEL (Business Process Execution Language [2]) is widely used for orchestrating web services. A business process (i.e. composite web service) exchanges messages with partner web services, considered as service providers. A business process is itself a web service with respect to its users. Web services have communication ports where operations are invoked. An unsuccessful operation gives rise to a fault.

The work in this paper incorporates several aspects of the component frameworks mentioned above. OSGi is used as the framework within which device services live. The approach reflects the principles of SOA. Although SCA has been evaluated as a flexible way of interconnecting home components [11], OSGi remains the main focus. As discussed in section III-B, the architectural concept of a filter is supported within the approach.

OSGi was originally conceived for use in a home environment, so it is hardly surprising that it is suitable for the author's purpose. However, several researchers have sought to extend its applicability. [9] enriches the SODA approach by dealing with data semantics. [8] focuses on self-configuration of home devices and personalisation of services offered to the user.

Sensor fusion has been studied as a general approach for sensor networks, but has found applications in home care. [5] describes middleware that is particularly designed for use in assisted living. [13] also address sensor fusion in home care, but emphasising the role of visual (camera) input.

The work in this paper generalises the sensor fusion idea significantly. Actuator fusion ought also to be supported, since it is often desirable to map one actuator output to many. Taking this idea further, it should also be possible to map between sensor inputs and actuator outputs in a flexible manner.

*4) Other Work:* The CRESS approach has affinities with other work. A CRESS feature is similar to the idea of a pointcut in aspect-oriented programming, which has been used for feature design [3]. Device services are also a form of pervasive/ubiquitous computing (e.g. [1]). However, context awareness is handled at the policy rather than device level.

### C. Structure of The Paper

Section II describes the methodology and notation of the service-oriented approach called CRESS (Communication Representation Employing Systematic Specification). The architecture and concepts of device services are explained in section III. Section IV illustrates the approach through examples of device services. Finally, section V rounds off the paper.

## II. CRESS

CRESS (Communication Representation Employing Systematic Specification, www.cs.stir.ac.uk/~kjt/research/cress.html) is a notation and set of tools for rigorous development of many kinds of services. For the work reported here, CRESS has been adapted to create device services.

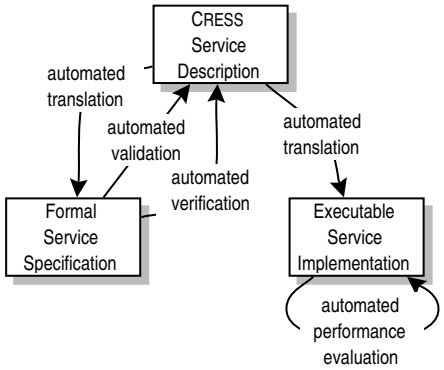


Fig. 1. CRESS Development Methodology

### A. CRESS Methodology

The general CRESS development methodology is illustrated in Fig. 1. The designer describes services using a graphical notation that is general-purpose and easy to learn. Although several graphical editors can be used, the preferred one is CHIVE (CRESS Home-grown Interactive Visual Editor, www.cs.stir.ac.uk/~kjt/software/graph/chive.html). This is well integrated with CRESS, e.g. the user can directly validate, verify and implement services from within the editor.

Having created a service description, the designer can then formally validate this. This is achieved through automatic translation to a formal specification, e.g. using the LOTOS standard (Language Of Temporal Ordering Specification). Formal validation is then performed using tests expressed in the MUSTARD language (Multiple-Use Scenario Testing And Refusal Description [14]). Validation is pragmatic – it can be performed quickly even if the state space of the specification is very large. For more precise analysis, the designer can also formally verify properties of a service. Again, this requires automatic generation of a formal specification. Verification is performed using properties expressed in the CLOVE language (CRESS Language-Oriented Verification Environment [16]).

Once confidence has been built in the service design, its implementation and deployment are automated. For web services, for example, a complex set of implementation files is generated. These make use of the standards for BPEL (Business Process Execution Language) and WSDL (Web Services Description Language). Although services are rigorously developed, issues such as performance may arise in an implementation. The designer is therefore able to evaluate the functional and non-functional correctness of a service implementation. This re-uses the *same* MUSTARD tests as were checked against the specification. The MINT tool (MUSTARD Interpreter) is also able to perform load tests and to check for consistency of performance.

BPEL is an effective way of orchestrating lower-level device services. However BPEL requires highly specialised knowledge, even using commercial design tools such as ActiveVOS Designer. CRESS creates code automatically for BPEL, but more importantly the high-level CRESS notation also allows automatic formal verification and validation.

### B. CRESS Notation

A CRESS diagram is a directed graph. In BPEL terms, this defines an executable business process. Someone who knows BPEL will find the CRESS representation familiar, but more compact. Numbered nodes (ellipses) contain actions (activities) for device services. Arcs between nodes may be labelled with expression guards or event guards. Expression guards decide which path is followed, dependent on some condition. Event guards introduce behaviour conditional on an event. Assignments may be used at the end of nodes and arcs in the form / *variable* <– *expression*.

CRESS names are in simple or hierarchic form. Operation names have the format *service.port.operation*. Simple variables have the types defined by XSD (XML Schema Definition, e.g. **Double** *d*, **String** *s*). Record types defined in braces {...} are accessed in the form *structure.field*.

Device services make extensive use of the predefined **Device** type. This is a structured type with fields *instance*, *period* and *params*. These are mapped to/from the event fields described in section III-A. As a convenient short-hand, fields of this type can be accessed using ‘ ’ ’ (e.g. ’*params* means *device.params*, while ’*instance3* means *device3.instance*).

The subset of CRESS activities appearing in this paper is explained below; CRESS supports much more than is described here. As usual, ‘?’ means optional.

**Device** *operation output input?* This is a syntactic convenience for calling a device service in the home (see **Invoke**), but also hides internal details such as device service naming.

**Empty** No action. The **Empty** label is usually omitted, leaving just an empty ellipse. This kind of node is useful for joining other nodes and for introducing local handlers.

**Finish** Used to indicate the end of a template diagram.

**Fork** Used to introduce parallel paths; further forks may be nested to any depth.

**Invoke** *operation output input?* Used for external services outside the home. An asynchronous (one-way) invocation sends only an output. A synchronous (two-way) invocation exchanges output and input with a partner service.

**Join** *condition?* Each **Fork** is matched by **Join**. By default, only one of the parallel paths leading to **Join** need terminate successfully. However, an explicit join condition may be defined over termination of parallel activities. This uses the node numbers of immediately prior activities. For example, 1&&(2||3) means that node 1 and either node 2 or 3 must terminate successfully. In turn, this means that their prior activities must also succeed.

**Receive** *operation input* Typically used at the start of a business process to receive a request for service. An initial **Receive** creates a new instance of the process, though a later **Receive** may be used for input from other services. An initial **Receive** is usually matched by a **Reply** for the same operation.

**Reply** *operation output* Typically used at the end of a business process to provide some output.

**Start** Used to indicate the first node of a diagram. It is normally omitted unless the initial node is ambiguous.

**Throw** *fault* Reports a fault to be caught by a handler.

The kinds of handlers used in this paper deal with events and faults. A handler can be global, scoped (applying to all nodes following an empty one), or local to a **Device**/**Invoke**.

**Catch** *fault* This introduces a separate flow for dealing with the specified fault.

**Timeout** *period* This introduces a separate flow for dealing with an alarm event.

Besides activity nodes, CRESS diagrams can also contain rule boxes (rounded rectangles) and connector labels (e.g. **Start**, **Finish**). Among other things, a rule box defines types, declarations, use of other diagrams, and macros. Thus:

**Uses** **String** reply / SPEECH
Surgery <- ″01786-832-210″

declares a variable *reply* of type **String**, and use of the SPEECH diagram. *Surgery* is a macro that expands to a telephone number. (Rule boxes also support a range of other capabilities not illustrated in this paper.)

## III. DEVICE SERVICES

The device services architecture is presented. Events are sent between ‘devices’ of all kinds (including services) and a policy server that handles high-level control of the home. An event transformer is used for flexible mapping of device events using graphically-defined logic.

### A. Device Service Architecture

The basic architecture followed in this work is that of OSGi. Components are therefore OSGi bundles that typically register services (e.g. for control of devices). This service-oriented approach makes it easy for components to use other components in a loosely coupled way. Services could, in principle, call each other directly. However, they are designed to communicate via an event bus (mediated by the OSGi Event Admin service). This further decouples components, allowing them to register only for events they are interested in.

The high-level device architecture is shown in Fig. 2. The home components are generically called ‘devices’, though this covers a variety of possibilities. Devices that provide inputs would conventionally be called sensors (e.g. medicine dispensers, motion detectors, video cameras). Devices that act on outputs would conventionally be called actuators (e.g. door locks, gas shut-off valves, video recorders). In a domestic setting, the term ‘appliance’ would also be used (e.g. CD player, microwave oven, TV). More significantly, ‘devices’ can also be software services (e.g. data logging, text messaging, weather forecasting).

Two components are distinguished in the architecture: the event transformer (low-level services that transform device events, see section III-B) and the policy server (high-level services that manage the home).

A variety of components have been developed for use in the home. Categories of home devices include appliances (e.g. DVD, fan, light, TV), communications facilities (e.g. email, messaging, speech input/output), environment monitoring (e.g. humidity, temperature, weather), multimodal interfaces (e.g.

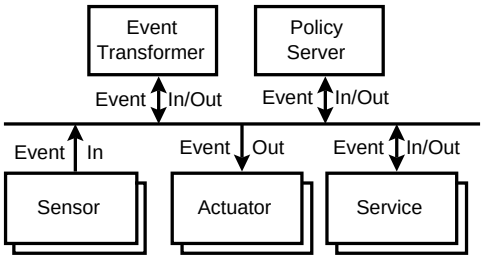


Fig. 2. Device Architecture

audio, gesture, touch, video), security (e.g. alarms, flooding, RFID), and telecare (e.g. bed-wetting, epileptic seizure, medicine dispenser).

Device input/output events (*device_in*, *device_out*) have a uniform structure with the following fields:

**message type:** the type of device input or output (e.g. *active*, *off*, *reading*).

**entity name:** the entity associated with a device message (e.g. *door*, *message*, *movement*). Some message types imply a unique entity (e.g. *system logger*), so the entity name can be omitted.

**entity instance:** the particular instance of an entity associated with a device message (e.g. *front*, *hall*, *outside*). Some entities may have only a single instance (e.g. *central heating system*), so the entity instance can be omitted. Entity instances may also identify groups (e.g. *all doors*, *upstairs windows*).

**message period:** the interval or time to which an event applies. For example, a temperature input would have period *15* if it was measured during the last 15 minutes, or *21:30* if it was measured at 9.30PM. For output, the same values could start a video recording in 15 minutes or at 9.30PM. The period is normally omitted, meaning 'now'.

**parameter values:** the device parameters. For example, this might give the reading for a temperature input, or the dimming percentage for a light output. The parameter values may be omitted if not relevant to the event.

### B. Event Transformation

In normal operation, devices cause input events that trigger the policy server. This results in actions that are sent to devices via output events. A home automation policy might be: 'when the front door is opened, turn on hall and lounge lights, play the user's favourite music, and activate climate control'. A telecare policy might be: 'when the user is late in taking medicine, speak a reminder in the relevant room; if the medicine is still not taken, alert a neighbour by text message'.

However, flexibility is desirable in handling device events. The event transformer can filter and modify events before they are seen by the policy server. Policies can be written to use the raw device events or the ones created by the event transformer. Possible patterns for event transformation include the following. Further combinations are possible, e.g. a service that reacts to an input event by generating both input and output events, or a service that ignores certain input events.

**in → in:** input events are mapped to input events. This is normally called sensor fusion, the idea being that raw input from several sensors can be combined to produce higher-level, synthetic triggers.

**out → out:** output events are mapped to output events. This should be termed actuator fusion, the idea being that synthetic actions can be mapped to several raw actuator outputs.

**in → out:** input events are mapped to output events. This supports low-level, device-oriented services (as opposed to the high-level, user-oriented services supported by goals and policies).

**out → in:** output events are mapped to input events. This lets policy actions trigger execution of further policies.

Event transformations could be coded in a conventional programming language. However, this would negate the goal of making it possible to change system functionality without detailed technical knowledge and programming. Event logic is therefore described using the CRESS design notation. This is automatically translated into BPEL for execution by a system that may be in the home or outside it.

Home components have an OSGi event interface, whereas BPEL processes have a web service interface. The event transformer therefore maps bidirectionally between OSGi events and web service calls. As a more precise description of the device architecture, Fig. 3 shows the relationship among the various components. Here, the event logic consists of BPEL processes created from CRESS diagrams. Note that events are normally handled by the event logic exclusively or not at all. With this usage, the event transformer sees only certain events and acts as a filter on these. However, it is sometimes useful for the event logic to process events that are also acted on by the policy server or a sensor/actuator service. This allows event logic to act as a monitor, and to supplement normal event processing with further events.

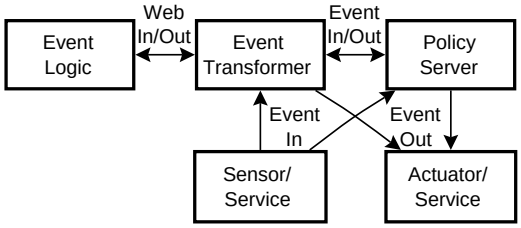


Fig. 3. Event Handling

The approach also offers a beneficial capability: device events can be handled by any external web service. This allows remote entities (e.g. a mobile phone or a PC) to control the home, and to receive information about significant home events (e.g. a low-temperature alarm or an intruder alert). Exposing home control to external entities is, of course, a risk so the web interface is secured.

## IV. SAMPLE DEVICE SERVICES

To illustrate the approach, a variety of small examples are given of device services. These have mainly been chosen to show some capabilities of the notation.

### A. CRESS Diagrams

CRESS supports three kinds of diagrams: configuration, root and feature. A configuration diagram is used to describe the

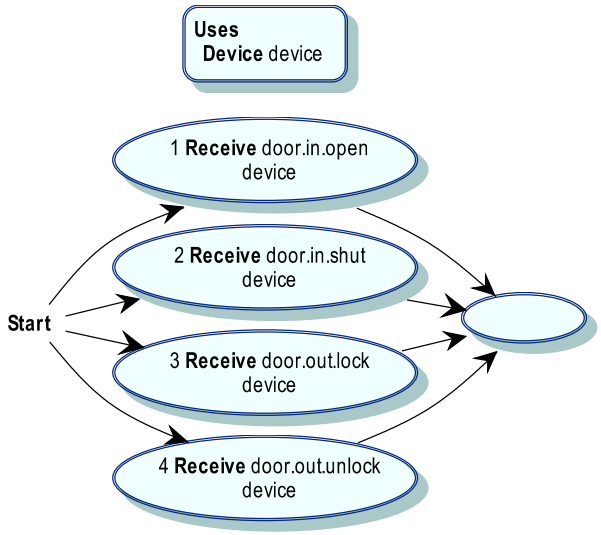


Fig. 4. Door Root Diagram

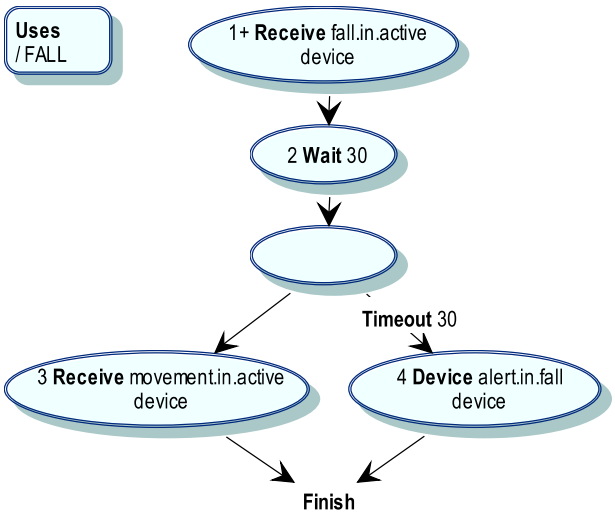


Fig. 5. Fall Detection Feature

services provided in some domain. For device services, this defines service characteristics. A configuration might state:

```
Deploys  -b2 -r .  /  DOOR DOOR_ALL DOOR_LIGHT
HOME   home    urn:Home    localhost:8181/axis
DOOR   door    urn:Door    localhost:8080/active-bpel
```

The first line gives deployment parameters ('-b2' means BPEL version 2, '-r .' means repeat all behaviour on termination) and the door services to be deployed. This is followed by one line per service that gives the service name, namespace prefix, namespace URI, and deployment URL. In this case, the HOME service (i.e. OSGi) is deployed on the local host at port 8181, while the DOOR service (including its features) is deployed on the local host at port 8080. In fact, services can be deployed on any system. In particular, device services can exist remotely and need not live on the OSGi home gateway.

CRESS supports the concept of feature that is common in other areas such as telephony. A root diagram describes the basic capability of a service. For device services, each class of device has its own root diagram. An example for doors is given in Fig. 4.

In this example, the rule box declares variable *device* of type **Device**. All device services make use of a *device* variable to convey device details, notably the instance (e.g. 'front' door, 'hall' light) and parameter values (e.g. heating temperature, spoken input). Features modify the root diagram, so they share the same variables. If one feature modifies *device* (e.g. to cause output to a different actuator), this must not interfere with other features. For this reason, it is normal for features to use their own copies of *device*.

The root behaviour of a door essentially lists all the input and output events associated with doors; at root level, no action is taken on these. Root diagrams for other devices (e.g. fall detection, speech) are similar in style. CRESS has two kinds of feature diagrams (spliced, template) that can be used to add functionality to a root diagram. A spliced feature defines a diagram that is effectively cut-and-pasted into the root diagram. This kind of feature tends not to be modular and so is not normally used. For device services, template diagrams are more appropriate. These add modular functionality to a root diagram by specifying triggering activities (i.e. events).

In a template feature, the triggering activity has the same form as in the root diagram (e.g. '**Receive** door.in.open'). A node number ending with '+' means that the feature is appended to the corresponding root node. (Note that root and feature node numbers need not match.) A template feature has a unique trigger node, and must have a unique final node (conventionally shown as **Finish**). All the remaining diagrams in this section are template features.

Feature interaction is a well-known problem in telephony, whereby different features can unexpectedly interfere with each other [4]. Care must therefore be taken over features for device services to ensure that this does not happen. For this reason, features typically define their own variables. Feature interaction is possible (and can be detected) in CRESS. Apart from a re-design to avoid interactions, CRESS also supports feature priorities to ensure predictable results.

### B. Fall Detection Feature (*in* → *in*)

Fig. 5 is a fall detection feature that adds to the basic FALL service. This performs sensor fusion by combining a raw fall signal with a movement signal to determine if a fall requires attention. Node 1 is a trigger for a fall detector message (reception of an 'active' fall input event). After a 30 second delay (node 2), an empty node introduces a receive activity (node 3) and an event handler. If a movement detector reports activity (node 3), it is assumed that the user has not had a serious fall. If there is no movement within the following 30 seconds, a fall alert event is generated (node 4). A **Device** activity like this communicates with the OSGi home platform). Both alternatives lead to the finish of the feature. (BPEL restrictions require the generated code to be more complex than the diagram suggests.)

### C. Door Locking Feature (*out* → *out*)

Fig. 6 is a feature that adds to the basic DOOR service, allowing all doors to be locked with a single policy action. This is an example of actuator fusion. If the target door is 'all', the

*device* value is copied to *device1* and *device2* (arc from node 1 to 2). The *lock* operation leads to parallel branches (node 2); parallelism avoids fixing on which door is locked first. One branch sets the *device1* instance to 'front' and requests that this door be locked, the other operates on *device2* for the 'back' door. Both parallel branches then join (node 5). The condition on this (3&&4) means that both nodes 3 and 4 must terminate successfully for the feature to complete successfully. Although not shown, a **Catch** fault handler could be introduced to deal with one or both *lock* operations failing.

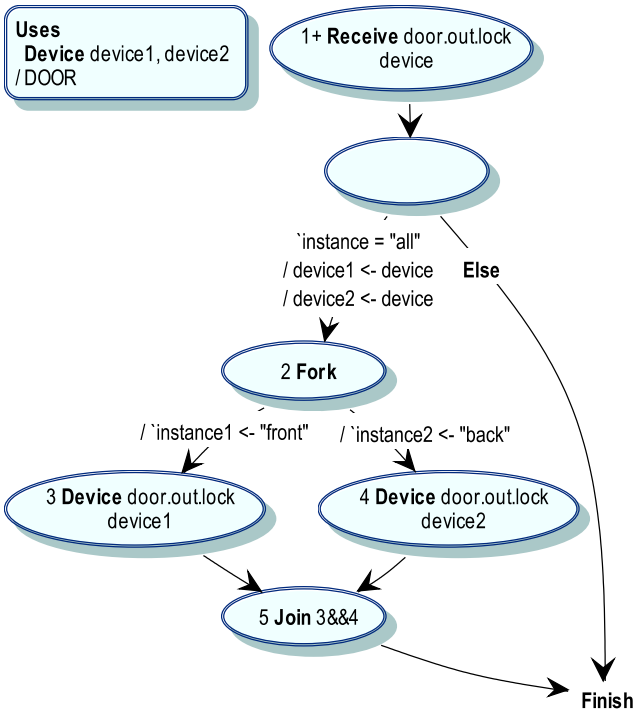


Fig. 6. Door Locking Feature

### D. Frost Warning Feature (out → in)

Fig. 7 is a feature that can trigger other policies when the heating is turned off in frosty conditions. If the heating is turned off (node 1), the outside temperature is read using *device1* as specification and *status* as the result (node 2). A *read* operation returns the most recent setting for a device (here, the outside temperature). In general a device status is a string, so it is converted to a numeric temperature in Celsius (assignment in node 2). If the temperature is 0 or less, a frost alert is generated. This is an input event to the policy server that can trigger further policies (e.g. to monitor the house plumbing for freezing conditions). This is an example of handling an action within the event logic as well as in another component: turning off heating is performed as normal by an actuator, but the event logic also reacts to this event.

### E. Entry Light Feature (in → out)

Fig. 8 is a feature that switches on an entry light when a door is opened at night. In fact, this (and the next example) could be defined as policies. If a service is low-level and device-oriented, it is more natural to specify it as a device feature.

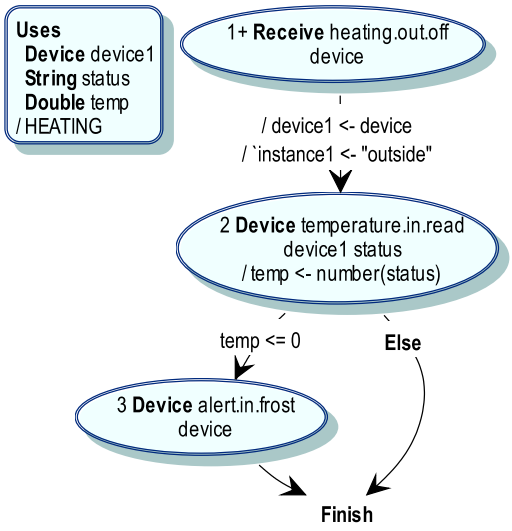


Fig. 7. Frost Warning Feature

If a service is high-level and user-oriented, it is preferable to specify it as a user-defined policy. The feature is triggered by a door being opened (node 1). An empty node then introduces a choice of paths. If the hour is between 10PM and 8AM, *device3* is set to *device*. A further empty node introduces checks on which door was opened: if the front door, then the hall light is turned on; if the back door, then the kitchen light is turned on; otherwise no action is taken.

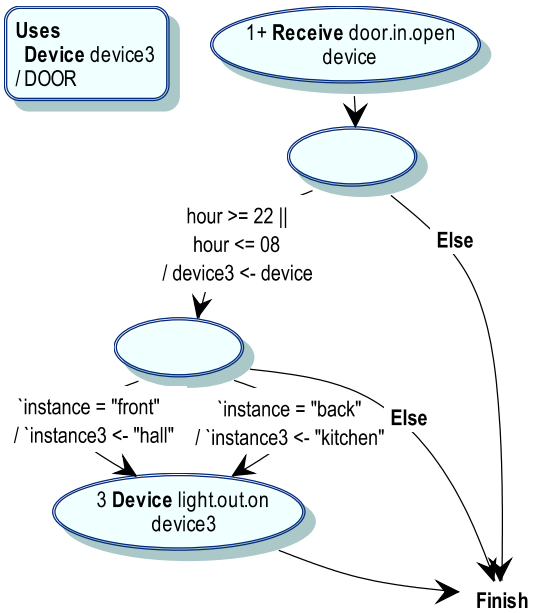


Fig. 8. Entry Light Feature

### F. Lounge Environment Feature (in → out)

Fig. 9 is a feature that, when someone opens the door to the lounge, turns on the light if necessary and sets the heating. Again, this is a feature that could be specified as a policy but would be less natural in that form. When a door is opened (node 1), it is checked whether it is the lounge door. If so, two parallel branches are followed (node 2); the parallelism here allows independent behaviour. The lounge light level is

read into *status*, and this is converted from a string into a numeric *level* (node 3). If this is less than 10% of normal, the lounge light is turned on (node 4). In parallel, the lounge heating is set for 21 Celsius (node 5). Because of the check on light level, node 4 may not be executed. As a result, only the second parallel path is required to succeed (as expressed by the join condition '5' in node 6).

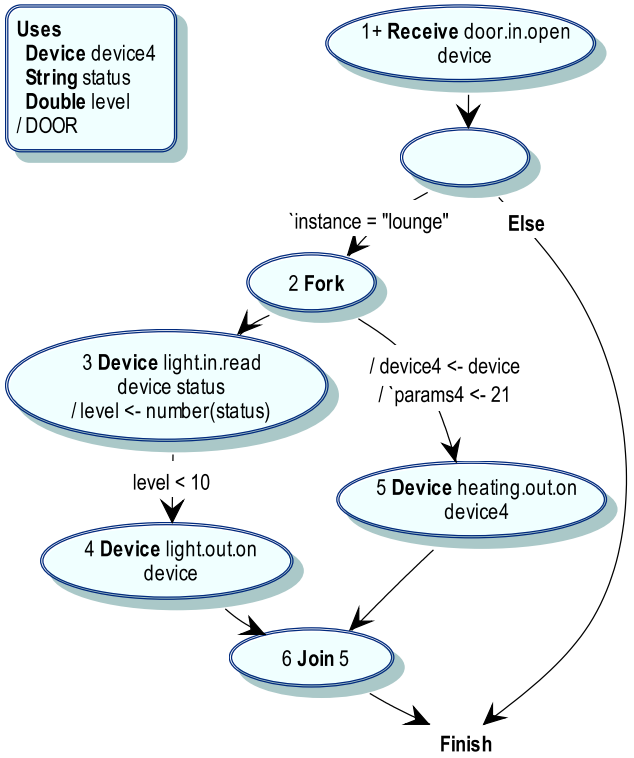


Fig. 9. Lounge Environment Feature

### G. Speech-Based Help Feature (in → out)

Fig. 10 is a more complex feature that provides speech-based advice to the user (Bob in this case). This feature relies on speech recognition and synthesis software from the University of Edinburgh. Because this feature supports a dialogue and uses an external web service, it could not be defined as a policy.

When the user makes a verbal request (node 1), it is checked if this is for the weather or for help; any other reply means the user is asked to repeat the request (node 9). For weather, an external weather forecast is requested for today's outlook (node 2). An **Invoke** like this causes interaction with an external service (and not the home platform). The outlook is then spoken to the user (node 3). For help, the user is queried as to the person who should be alerted (node 5). If a friend, a text message is sent asking for a call to Bob (node 6). If the doctor, a synthesised message is sent to the surgery reporting that Bob is ill (node 7). If the user does not specify either person, a general alarm input event is raised for action by the policy system (node 8).

### H. Realising Services

As noted in section II-A, device services are automatically verified, validated and realised. These aspects can be explained only briefly here.

Verification is achieved by automatically translating CRESS diagrams into LOTOS and then model-checking desirable properties. Normally this would require highly specialised knowledge of logics and tools ($\mu$-calculus and CADP for LOTOS). Instead, CRESS makes verification accessible through the high-level approach of CLOVE. This can verify a wide range of properties: freedom from deadlock, livelock and starvation; safety and liveness; and service-specific characteristics. As a small example, the following property must hold for Fig. 10: if a weather forecast is requested, this must eventually be spoken (a string here). As a global property, this response must be obtained in all behaviours of the service. Strings are preceded by '′', and any value of a type by '?'.

```
property(Weather_Response,
  response(global,
    signal(speech.in.text,′weather),
    signal(speech.out.text,?string)))
```

Verification can be time-consuming and impracticable for complex services. Formal validation using MUSTARD is also offered as a practical alternative. The following small example is an acceptance test that checks the service in Fig. 9. Initially a door open event is sent for the lounge. One parallel branch reads a request for the lounge light level, sends back a response of 8.3%, and expects a request for the lounge light to be turned on. The other parallel branch expects a request for the lounge heating to be set to 21 degrees.

```
test(Lounge_Entry_Low_Light,
  succeed(
    send(door.in.open, Device(′lounge,?Text,?Text)),
    interleave(
      sequence(
        read(light.in.read, Device(′lounge,?Text,?Text)),
        send(light.in.read, ′8.3)),
        read(light.out.on, Device(′lounge,?Text,?Text))),
      read(heating.out.on, Device(′lounge,?Text,′21)))))
```

As well as being automatically formalised, CRESS diagrams are automatically compiled into code (BPEL, WSDL and deployment descriptors for device services). These are automatically deployed for execution by the ActiveBPEL engine (www.activebpel.org). MUSTARD scenarios are then used to evaluate implementation performance under load conditions.

## V. CONCLUSION

It has been argued that the concept of sensor fusion should be extended for flexible support of device inputs and outputs. This permits actuator fusion, and also allows low-level, device-oriented services to be defined. These complement the high-level, user-oriented services supported by goals and policies. Although a feature or policy might sometimes be used for the same purpose, in practice the choice of which is usually clear.

OSGi input/output events are mapped to/from BPEL calls. A separate BPEL engine reacts to these events, providing device services to the home platform. Device services can map input/output events in all combinations.

Event logic is described using CRESS. This provides a simple graphical notation that is automatically formalised, validated, verified and implemented. This supports flexible management of smart homes for both home automation and

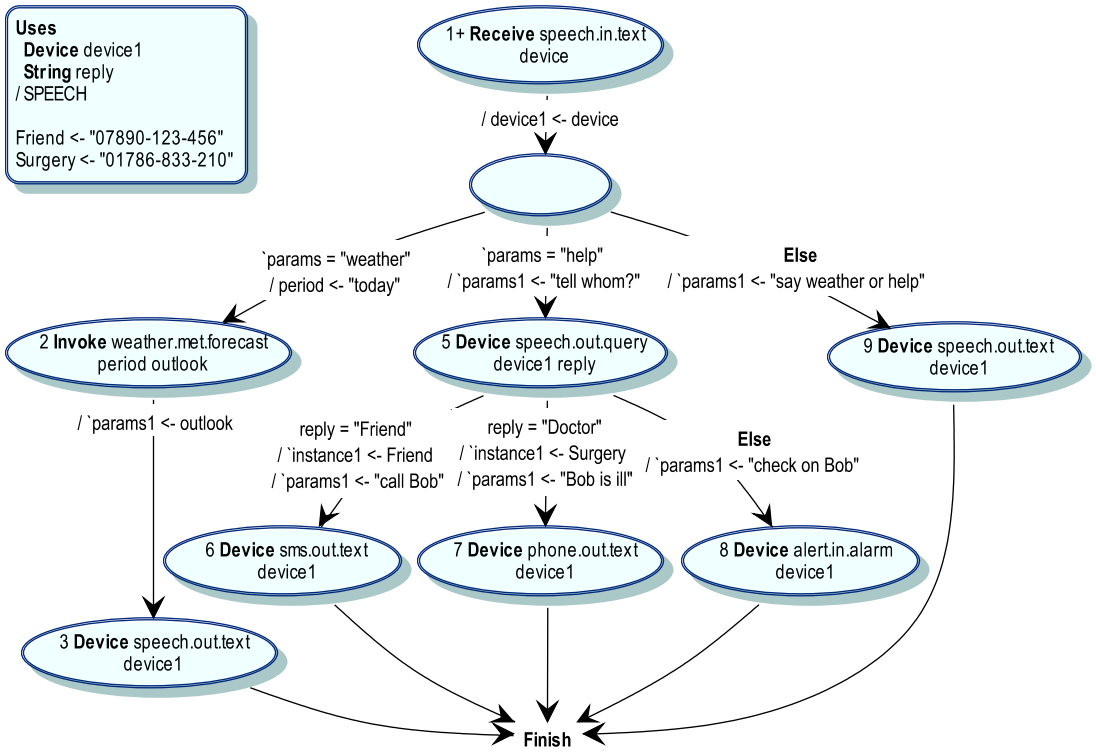


Fig. 10. Speech-Based Help Feature

telecare. Detailed technical knowledge and programming skills are not required to create device services. Although it is intended that end users (and their carers) define policies, creating device services is more appropriate for technicians who manage devices in the home.

The entire home system described in this paper has been evaluated in a lab setting, and also in a typical home. The response time of a device service is around one second. This offers adequate performance since only significant device events are considered (e.g. open, shut) and not frequent device signals (e.g. every movement). An extended trial of the whole approach is about to start in homes operated by Midlothian Council in Scotland.

## Acknowledgements

The author thanks Claire Maternaghan (University of Stirling) who wrote a number of the device bundles. The work was conducted in the context of the MATCH project (Mobilising Advanced Technologies for Care at Home, www.match-project.org.uk, supported by the Scottish Funding Council under grant HR04016).

## References

[1] I. Amundson *et al.*. OASiS: A service-oriented middleware for pervasive ambient-aware sensor networks. Technical Report ISIS-06-0706, Vanderbilt University, 2006.

[2] A. Arkin *et al.*, editors. *Web Services Business Process Execution Language*. Version 2.0. OASIS, Billerica, Apr. 2007.

[3] L. Blair and J. Pang. Feature interactions – Life beyond traditional telephony. In M. H. Calder and E. H. Magill, editors, *Proc. 6th Feature Interactions in Telecommunications and Software Systems*, pp. 83–93. IOS Press, May 2000.

[4] E. J. Cameron *et al.*. A feature-interaction benchmark for IN and beyond. *IEEE Comms. Magazine*, 31(8):18–23, Aug. 1993.

[5] L. Coyle *et al.*. Sensor fusion-based middleware for assisted living. In C. Nugent and J. C. Augusto, editors, *Proc. 4th Int. Conf. on Smart Homes and Health Telematics*, pp. 281–288. IOS Press, Jun. 2006.

[6] S. de Deugd *et al.*. SODA: Service-oriented device architecture. *Pervasive Computing*, 5(3):94–98, 2006.

[7] L. A. Gavrilov and P. Heuveline. Aging of population. In P. Demeny and G. McNicoll, editors, *The Encyclopedia of Population*, pp. 27–50. MacMillan, Jan. 2003.

[8] P. Gouvas, T. Bouras, and G. Mentzas. An OSGi-based semantic service-oriented device architecture. In R. Meersman, Z. Tari, and P. Herrero, editors, *Proc. On the Move to Meaningful Internet Systems*, LNCS 4806, pp. 773–782. Springer, Nov. 2007.

[9] J. E. López de Vergara *et al.*. An autonomic approach to offer services in OSGi-based home gateways. *Computer Communications*, 31(13):3049–3058, Aug. 2008.

[10] B. Margolis and J. L. Sharpe. *SOA for The Business Developer*. MC Press, 2007.

[11] C. Maternaghan and K. J. Turner. A component framework for telecare and home automation. In *Proc. 7th Consumer Communications and Networking Conference*. IEEE Press, Jan. 2010.

[12] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In *Proc. 6th. European Software Engineering Conference/Proc. 5th. Symp. on the Foundations of Software Engineering*, pp. 60–76, Zurich, Sep. 1997.

[13] A. M. Tabar, A. Keshavarz, and H. Aghajan. Smart home care network using sensor fusion and distributed vision-based reasoning. In R. Cucchiara, J. K. Aggarwal, and A. Prati, editors, *Proc. 4th Int. Workshop on Video Surveillance and Sensor Networks*, pp. 145–154. ACM Press, Oct. 2006.

[14] K. J. Turner. Validating feature-based specifications. *Software Practice and Experience*, 36(10):999–1027, Aug. 2006.

[15] K. J. Turner and G. A. Campbell. Goals for telecare networks. In A. Obaid, editor, *Proc. 9th Int. Conf. on New Technologies for Distributed Systems*, pp. 270–275. Montreal, Jul. 2009.

[16] K. J. Turner and K. L. L. Tan. A rigorous methodology for composing services. In M. Alpuente, B. Cook, and C. Joubert, editors, *Proc. Formal Methods for Industrial Critical Systems 14*, LNCS 5825, pp. 165–180. Springer, Nov. 2009.

[17] F. Wang and K. J. Turner. Towards personalised home care systems. In I. Maglogiannis, editor, *Proc. 1st Int. Conf. on Pervasive Technologies related to Assistive Environments*, pp. L2.1–L2.7, ACM Press, Jul. 2008.