PROGRAMMING THE SMART HOME

Urs Bischoff, Vasughi Sundramoorthy, Gerd Kortuem

Lancaster University, Computing Department, UK

Keywords: Smart home, intelligent environment, programmable space, software engineering.

Abstract

A smart home is a house that is responsive to its inhabitants and their actions by being aware of their context. Potential applications for smart homes address economic and comfort aspects of living, or could provide unobtrusive support for the elderly or disabled to promote independent living. The basic building blocks of such a smart home is a computing system consisting of distributed sensors and actuators. Programming and maintaining such an infrastructure is challenging because suitable programming abstractions are currently missing. In this paper we introduce the notion of programmable space that lets the application developer perceive the smart home as an integrated runtime environment. This approach is implemented in a system called RuleCaster. Applications are developed in a high-level rule-based language. Our approach shows a notable simplification of application development and maintenance. To verify the utility of RuleCaster we use a scenario-based evaluation method.

1 Introduction

Smart homes are an important area of research for building ambient intelligence and intelligent environments. A smart home is a domestic environment in which we are surrounded by interconnected technologies that are, more or less, responsive to our presence and actions [8]. With the advent of better sensors, tiny low-powered computers and wireless communication, we are moving closer to realising this vision of an unobtrusive pervasive computer assistant. This area is interesting because we tend to spend more time in our homes than in other environments.

Smart home applications can be divided into two broad classes [2]. The first class is related to economic and comfort aspects of living. For example, window blinds automatically adjust to satisfy the light and temperature requirements depending on the location and activities of the inhabitants. The second class addresses independent living. Assistive technology for supporting the daily lives of the elderly is one example.

Regardless of the application the basic building block of a smart home is a network consisting of a large number of heterogeneous sensor and actuator nodes. These nodes are small low-powered computers populated with sensors and/or actuators that allow the computing system to interact with the surrounding environment. As the technology makes the creation of these smart environments feasible, we must also consider the way we program and maintain the underlying infrastructure.

A problem pointed out in [10] is that our experience in building integrated environments is limited by the set of concepts we know at the time of development; we are constantly faced with better technology becoming available. Unlike a mobile phone it is not possible to replace the smart home every few months. As emphasised by Rodden and Benford [16], living and work environments are also subject to continuous transformation; they are modified by the people who inhabit them in a variety of ways, for a variety of purposes and with different frequencies. This observation also has an effect on the requirements of a computing infrastructure and its applications. As such, they are subject to similar changes as the rest of our living and work environments. As users we want to change or extend the application logic from time to time. With better technology becoming available, the network or parts of it should be exchanged without greatly affecting the running applications. Or by adding nodes and replicating tasks we want to make the application more reliable. Upgrading and modifying embedded software is difficult because of the generally tight coupling between hardware and software. Having to deal with distributed applications makes changes to the application even harder.

In this paper we present RuleCaster, a programming system for the smart home. This system supports evolutionary changes in the life-cycle of an application in a unified way by separating the concerns of implementation, distribution and execution of an application. Furthermore, this high-level approach relieves the programmer of dealing with low-level details such as sending data between nodes, and issues related to distributed programming. The application developer defines applications in a high-level language for the smart home as a whole. A compiler then splits the application into a set of distributed tasks that form the executable application code. The novelty of our approach is that the programmer perceives the smart home as an integrated runtime environment. Space is a fundamental concept of the physical environment of a smart home. The kitchen, for example, is a space that has different characteristics to the bedroom. RuleCaster directly reflects the notion of physical space in software in terms of a `space` data type. The programmer is presented with high-level software abstractions of spaces that allows him to program the smart home as if it was a single computing platform.

This paper is organised as follows. In the following section we give a brief overview of the area of smart homes. In Section 3 we introduce a concrete example of a smart home

and several application scenarios that are used to illustrate our approach in the remainder of this paper. In Section 4 we present a solution outline. Then we focus on one specific part, namely a high-level application language, of the overall approach. Section 5 presents a software abstraction that builds the basis for the application language introduced in Section 6. In Section 7 we show the utility of our approach with a scenario-based evaluation. Finally, Section 8 concludes this paper.

2 Background on Smart Homes

The development of small low-powered computers, sensors and wireless radios has made impressive progress in recent years. This is only partly the consequence of Moore's law and better materials being developed in research labs. The potential economic impact of this technology (not just for the smart home, but in general) shows great potential beyond research labs (e.g. [9]). Several companies produce ready to use hardware platforms (e.g. Crossbow, Ember, Freescale or Texas Instruments) and industry-driven communication standards have been published (e.g. Zigbee [1]).

This technological development and the trend of moving away from a single PC per household has cleared the way to the development of smart environments. Several research groups have developed first-generation prototypes of the smart home to study the computing needs in our everyday lives ([15, 12, 11]).

Currently, the main technological focusses are on integrating suitable hardware into the home, and on developing services that analyse sensory data in order to identify high-level contexts. One existing problem is that these systems are generally purpose-built. This makes the development or maintenance of applications a challenging task because expert knowledge is required.

In order to ease the development of applications for these systems, people have developed middleware solutions and programming abstractions. The context toolkit [7] provides the application developer with *context widgets* that give access to context information while hiding the details of context sensing. These widgets can be used in a programming language to implement context-aware applications.

Similarly Helal et al. propose a service-based middleware to simplify application development [11]. Basic services hide low-level details of a specific sensor platform by providing a standardised interface to the application developer. These services can be composed into other services or be directly used in a development environment to build smart home applications. The underlying middleware takes care of the management of the distributed infrastructure.

Still, the application programmer is faced with the problem of having to decompose the global application logic into a set of distributed tasks. Furthermore, such a static distribution is not flexible for accommodating future changes. In contrast to these bottom-up approaches that allow the application developer to build higher level abstractions from low-level services, we propose a top-down approach. Our approach provides abstractions for the smart home as one logical entity instead of individual devices. Hence, the programmer is not forced to decompose the application logic into distributed tasks and implement them by composing device-centric services. Instead, applications can be directly written for the smart home as a whole.

3 The Off-the-shelf Smart Home

In order to simplify the discussion in the remainder of this paper we introduce concrete application scenarios. First, we show the smart home infrastructure in Section 3.1. Then, we describe the scenarios in Section 3.2.

3.1 Infrastructure

The smart home is equipped with a variety of sensors and actuators. Just like electric wiring or central heating, these devices are part of our infrastructure. Figure 1 shows a smart home that is equipped with off-the-shelf hardware. The home is subdivided into four rooms: kitchen, bedroom, office and corridor. Each room contains a number of sensors and actuator that build the interface between the computing system and the physical world:

Kitchen. The *motion sensor* detects the presence of moving objects. The *smart stove* senses if it is turned on or off. And the *automatic window* can open and close itself.

Bedroom. It contains an *automatic window*. The *automatic blinds* allow to be automatically adjusted. The *smart bed* has integrated pressure sensors to detect the presence of a person on the bed.

Office. The *motion sensor* detects the presence of moving objects. The *ambient display* is an unobtrusive screen that can display arbitrary RGB colours.

Corridor. The *alarm* can alert inhabitants with sound and light effects. The *smart mail box* detects mail. The *motion sensor* detects the presence of moving objects.

Furthermore, every room contains a *switch/sensor* that is used to turn *on/off* the light and to detect if the corresponding lamp is turned on or off.

3.2 Application Scenarios

Alice is the proud new owner of the smart home introduced in Section 3.1. She has put a lot of effort and money into adapting her living space to her taste and needs. Similarly, she also wants to use her computer infrastructure to improve her living comfort and safety. Because of previous experience with forgetting to turn off the stove she wants the alarm to be turned on if the stove is switched on when nobody is in the kitchen. Meanwhile Alice often works from home. Mail is delivered to her house several times a day. She wants to be notified as soon as mail arrives when she is at work in her home office. During the night she feels more protected if the blinds are shut; hence, she wants the blinds to be automatically shut when she is in bed and has turned off the light. As an energy conscious person she also knows that shut blinds provide additional insulation.

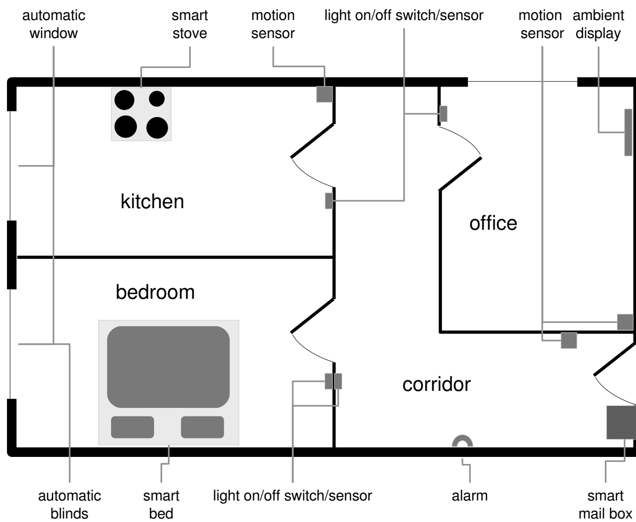

Figure 1: The programmable smart home.

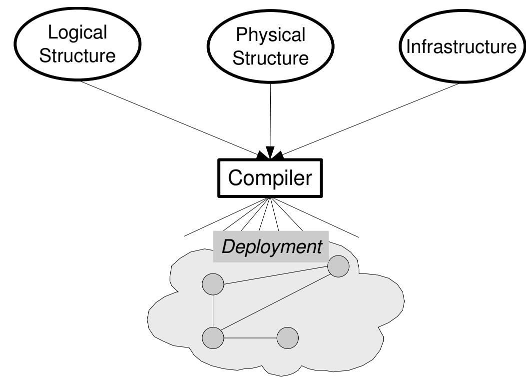

Figure 2: Changes to one of the three models can be directly propagated by re-compilation.

4 Solution Outline

In the following we outline our solution that addresses the two main problems mentioned earlier: (1) defining the global application in terms of local actions, and (2) accommodating changes. By dealing with space not only as a physical entity but also as a software data structure we realised the concept of a programmable physical environment. The advantage of this approach is that the application developer can implement applications for the smart home as one logical entity; *the smart home is the computer*. RuleCaster provides the application developer with a high-level language for application development.

As mentioned earlier, dealing with evolutionary changes to the application requirements and the infrastructure is a complex job for application developers, which touches upon several life-cycle phases. We can identify three major classes of changes that a smart home application undergoes throughout its lifecycle. These are changes to the

- logical structure,
- physical structure and
- computing infrastructure.

The logical structure describes how functional elements and application states are connected for describing the application logic. Changes to the logical structure refer to changes in the observable behaviour of an application. For example, while an application might initially be defined to open the window when the room is too hot, a new application logic might turn on the air conditioning instead.

The physical structure describes where computational elements are executed and where application states are stored in the network infrastructure. Changes to the physical structure of an application refer to changes in the distribution of computing tasks to individual nodes. For example, a task initially performed by a central node is distributed over several nodes in order to improve reliability and decrease energy consumption.

The infrastructure is the actual smart home that stores and computes the application states. It is the distributed runtime environment. Changes to the infrastructure refer to changes in the underlying hardware and runtime system (e.g. network system). For example, a system might need to be updated when a new generation of hardware devices becomes available with different processor, memory or radio. Another example is modifying a system by adding or removing nodes.

RuleCaster supports these classes of changes by separating them into three separate models (cf. Figure 2). The core of RuleCaster is a high-level compiler system that takes these three models as input to generate a distributed software system. Changes to any of these three models can be directly propagated to the running application by recompilation and re-deployment. This separation allows the programmer to change any model individually and therefore simplifies maintenance of applications.

The infrastructure is described in terms of a network model which provides the compiler with a list of properties (e.g. location, hardware specification etc.) and available node-level services. The infrastructure consists of the actual sensor-actuator node hardware running a middleware that executes the application. This middleware is based around a service-based architecture. Services give access to the interface between the network and the physical world (i.e. sensors and actuators). Nodes are statically assigned to a space or several spaces (e.g. a node in the kitchen is assigned to the space *kitchen*).

In this paper we exclusively focus on the the description of the logical structure of an application (or application logic). More information about the other models can be found previous work [3, 4, 5]. The application logic is defined in the RuleCaster Application Language (RCAL) — a programming language that provides the application developer with high-level abstractions of the programmable smart home.

Figure 3 depicts the architecture of the implemented system. The structure of the programming model is reflected in this architecture. The application, which is programmed in the high-level language RCAL is split by the compiler into individual tasks and distributed over the network of sensor nodes. The physical structure is implicitly described by a

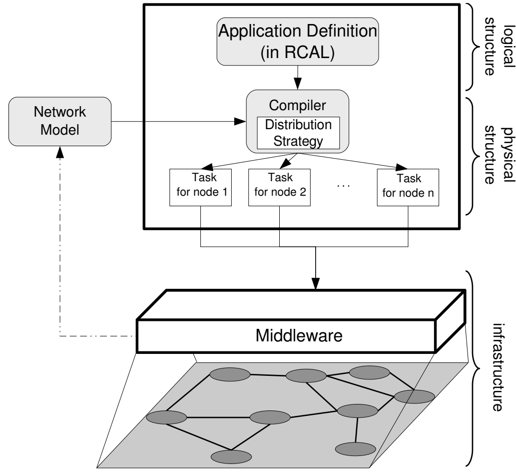

Figure 3: The architecture of the RuleCaster system.

distribution strategy — a compiler plug-in that influences the generation of the physical structure of an application based on some quality model such as energy consumption or robustness.

5 The *Space* Programming Approach

The smart home is an integrated runtime environment that can execute user-defined applications. We propose a high-level language for implementing applications. The basic data type of this language is `space`. A `space` is an abstract representation of a distributed computer consisting of a collection of nodes with some relationship to one another such as "all nodes in the same room" or "all nodes within a circle of radius r around position X". A node can belong to one or several spaces. Even though the `space` type could be used to define an arbitrary collection of nodes, we solely use it to describe a network partition contained within a confined spatial area.

The `space` data type hides low-level details of the underlying network such as the number of nodes or specific communication protocols and encapsulates the manipulation of state information. Figure 4 illustrates the `space` data type. This data type has four distinct communication interfaces. It uses sensors to observe and actuators to influence the surrounding physical world. Furthermore, it communicates with other spaces through the exchange of `space` state information.

The smart home is represented as a set of programmable spaces. Each `space` encapsulates the state of the corresponding network partition. The application developer implements applications in terms of these states and conditions for state transitions. These conditions can depend on sensor observations, the activation of actuators or on the state of other spaces (i.e. it requires state information; cf. Figure 4). By providing state information to other spaces or requiring state information from other spaces, the spaces create an implicit coordination network between each other.

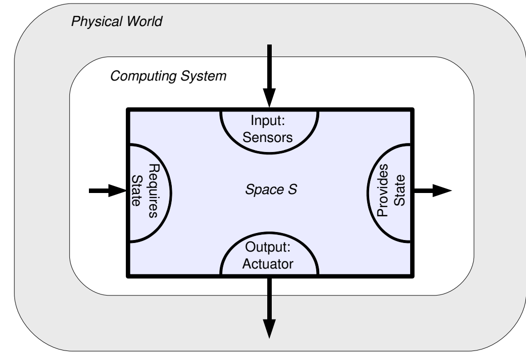

Figure 4: The data type `space` provides the application developer with a high-level abstraction of programmable space.

```
PRE_STATE(<state1>) [
     <rule>.
] POST_STATES(<state2>,...,<staten>).
```

Figure 5: The basic structure of a state transition rule.

5.1 A Rule-based Language

As mentioned above the space data type encapsulate the application logic that corresponds to the respective physical space. The rule-based language RuleCaster Application Language (RCAL) is used to describe states and the conditions for state transitions. Figure 5 shows the basic structure of a state transition rule. The programmer specifies the conditions for the transition from one current state (labelled with `PRE_STATE`) to a set of next states (labelled with `POST_STATES`) in a rule. This rule is a boolean function consisting of disjunctions and conjunctions of boolean terms. Boolean terms are defined in terms of built-in functions (e.g. comparison functions), user-defined functions, actuator activation functions or filter functions of sensor observations. For example, a system which is in a state `cold` can switch into states `hot` and `humid` when the sensors are observing high temperature and high humidity.
The advantage of this high-level language approach is that rules can be easily understood by both humans and computers; Dey et al. even argue that people are naturally inclined to use rules when asked to describe the behaviour of a smart space [6]. Furthermore, the declarative nature of RCAL does not force the programmer to express where and how the application is executed. He only has to specify what should be executed by the smart home.

6 The RuleCaster Application Language

We illustrate the syntax and expressiveness of RCAL with the example scenarios introduced in Section 3.2. We use the following two expressions in the description of the language syntax:

Definition 6.1. *A* term *is a boolean function that is either* `TRUE` *or* `FALSE` *at any point in time. It has an ID and zero or more arguments. An argument is either a value that*

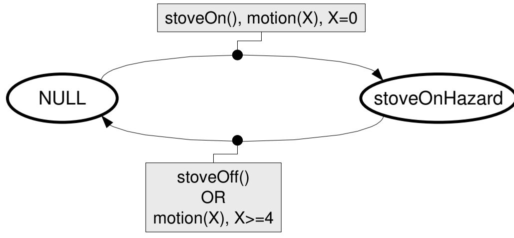

Figure 6: The kitchen states and the conditions for the state transitions. `Null` represents the empty default state.

is provided to the function or one that is returned from the function.

Definition 6.2. *The* signature *of a term is defined as a tuple A/x, where A is the ID of the term and x is the number of arguments.*

In the following we show how these scenarios are expressed in RCAL in terms of spaces, states and state transitions. The four rooms (kitchen, bedroom, corridor and office) are the four spaces we program. The user-specified rules are written in bold italics followed by the explanation of the implementation in RCAL.

Turn on the alarm if the stove is switched on and nobody is in the kitchen. This scenario includes the spaces `kitchen` and `corridor`. Figure 7 illustrates the RCAL program that implements the application logic. Line 1 introduces the declaration of the space `kitchen`. Lines 2-6 declare the communication interface of this space (cf. Figure 4). It has three sensors[1] to observe the environment.

The sensor is declared with the signature of a term that can be used in a transition rule to access the sensor. Line 3, for example, defines the term with ID `stoveOn` that requires `0` arguments. In the following the term `stoveOn()` can be used in transition rules; this term is `TRUE` if the stove is detected to be on and `FALSE` otherwise. The space `kitchen` provides information about the state `stoveOnHazard` to other spaces (declared in line 6).

Figure 6 depicts the two kitchen states and the transition conditions used to implement the application logic for the space `kitchen`. The first state transition is defined in lines 8-10. The transition is to the state called *stoveOnHazard*; this state describes the situation when the stove is turned on and nobody is in the kitchen. Because `PRE_STATE` is empty, the transition generates a new state. The actual condition for the state transition is defined in line 9. It is defined as the conjunction of the three terms `stoveOn()`, `motion(X)` and `X=0`. It is important to notice that both terms `stoveOn()` and `motion(X)` have the same signature as the corresponding sensors in the interface declaration; this allows the use of sensor observations in a rule. The argument in `motion(X)` is term specific; this term is true if the current sensor specific motion value can be assigned to `X`. The last term compares the value that is assigned to `X` with `0`. If the conjunction of these three terms

[1] Such a sensor is an abstract concept of programmable space and does not directly correspond to a specific physical network sensor.

```
1     SPACE(kitchen) {
2       INTERFACE:
3         SENSOR(stoveOn/0),
4         SENSOR(stoveOff/0),
5         SENSOR(motion/1),
6         STATE(stoveOnHazard).
7
8       PRE_STATE() : time(2) [
9         STATE :- stoveOn(), motion(X), X=0.
10      ] POST_STATES(stoveOnHazard).
11
12      PRE_STATE(stoveOnHazard) [
13        STATE :- stoveOff().
14        STATE :- motion(X), X>=4.
15      ] POST_STATES().
16    }
17
18    SPACE(corridor) {
19      INTERFACE:
20        SENSOR(mail/0),
21        ACTUATOR(alarm/1).
22
23      PRE_STATE() : STATE(kitchen:stoveOnHazard) [
24        STATE :- alarm(10).
25      ] POST_STATES(alarmOn).
26
27      PRE_STATE(alarmOn) : STATE(NOT kitchen:stoveOnHazard) [
28        STATE :- alarm(0).
29      ] POST_STATES().
30    }
```

Figure 7: RCAL program for the programmable smart home.

is `TRUE`, the condition for a state transition is satisfied (i.e. the stove is on and nobody is in the kitchen because the measured motion is zero). There is an additional temporal constraint for the successful evaluation of this condition. Line 8 defines `time(2)`; this means that all terms have to be `TRUE` at arbitrary points within a common interval of 2 seconds.

Lines 12-15 specify the transition from the state `stoveOnHazard` to the empty state. The transition condition is satisfied if either the condition in line 13 or the one in line 14 is satisfied. In other words, the state `stoveOnHazard` is removed if the stove is turned off (i.e. `stoveOff()` is `TRUE`) or if someone is in the kitchen (`motion(X), X`$\geqslant$`4` is `TRUE`; the higher the value assigned to `X` the more probable is the presence of a person in the kitchen). The order of the two rules is of no significance with respect to the evaluation order of these rules.

Lines 18-30 define the space `corridor`. This space is responsible for triggering the alarm in case of a hazardous situation in the kitchen. This space has two communication channels in its interface declaration; it is the actuator with ID `alarm` and the sensor with ID `mail`. Lines 23-25 define the transition from the empty state to the state `alarmOn` (i.e. the valid state when the alarm is turned on). If the evaluation of the term `alarm(10)` in line 24 is *TRUE*, the condition for this transition is satisfied. Similar to the temporal constraint line 8 there is a state constraint for this transition defined in line 23 as `STATE(kitchen:stoveOnHazard)`. This means that the space kitchen has to be in the state `stoveOnHazard` in order to satisfy the transition condition. This state constraint concept allows the coordination between different spaces. The space `corridor` has access to internal state information of the space `kitchen` because this state is declared in line 6 of the definition of the space `kitchen`.

The transition from the state `alarmOn` to the empty state is defined in lines 27-29. The state constraint declared

```
1     SPACE(bedroom) {
2       INTERFACE:
3         SENSOR(bed/1),
4         SENSOR(isLightOn/0),
5         SENSOR(isLightOff/0),
6         ACTUATOR(blinds/1).
7
8       PRE_STATE() [
9         STATE :- bed(X), X>20.
10      ] POST_STATES(personInBed).
11
12      PRE_STATE(personInBed) [
13        STATE :- isLightOff(), blinds(0).
14      ] POST_STATES(personInBed, blindsDown).
15    }
16
17    SPACE(office) {
18      INTERFACE:
19        SENSOR(floor/1),
20        ACTUATOR(display/3).
21
22      PRE_STATE() : STATE(corridor:newMail) [
23        STATE :- officeOccupied(), display(255,0,0).
24        officeOccupied() :- motion(X), X>=4.
25      ] POST_STATES(mailAlert).
26    }
```

Figure 8: Definitions of the spaces office and corridor.

in line 27 requires the `kitchen` not to be in the state `stoveOnHazard`. The transition condition is satisfied if the term `alarm(0)` is `TRUE` (i.e. the alarm can be turned off by the corresponding actuator).

Shut the blinds if someone is in bed and the light is turned off. Figure 8 shows the definition of the space `bedroom`. For simplicity reasons we only show the most important transitions. In line 9 the weight of the object on the bed is assigned to `X`. If this value is more than `20`, the transition condition is satisfied and the bedroom goes into a state `personInBed` (i.e. it is assumed that a person is in bed). Lines 12-14 show the concept of a transition from one state to several states. This allows us to keep the state `personInBed` and add the additional state `blindsDown` to the current states of the space `bedroom`.

Notify the person in the office as soon as mail arrives. Lines 17-26 in Figure 8 define the space `office`. In order for the transition rule defined in lines 22-25 to work we have to extend the definition of space `corridor` shown in Figure 7. The space `office` requires access to information about the state `newMail` (i.e. the state that is valid if mail is in the mail box) in space `corridor`. Access from outside is granted by adding the statement `STATE(newMail)` to the interface declaration of the space `corridor`. The transition rule to state `newMail` in space `corridor` is shown in Figure 9.

Lines 23 and 24 in Figure 8 show another important concept of RCAL. The term `officeOccupied()` in line 23 is defined with the rule in line 24. This rule is satisfied if the conjunction of both its terms (`motion(X)` and `X`$\geqslant$`4`) evaluates to `TRUE`. It is also important to notice that a lazy evaluation semantic is applied. This is a powerful concept that can be used to influence the output of information through actuators. For example, the actuator term `display(255,0,0)`, that causes the display to show the colour red, in line 23 is not evaluated if the term `officeOccupied()` evaluates to `FALSE`.

```
PRE_STATE() [
  STATE :- mail().
] POST_STATES(newMail).
```

Figure 9: Transition rule to the state *newMail* in space *corridor*.

7 Scenario-based Evaluation

One main challenge of developing smart home applications are changing application requirements (cf. Section 1). A problem faced is the fact that our experience in building integrated environments is limited by the set of concepts we know at the time of development. Once built for a specific goal (such as to assist the elderly or avoiding hazardous situations), the smart home will likely to be used for decades to come [10]. Still, we cannot assume that the application requirements or the physical environment will be static. RuleCaster addresses these changes by separating the development of applications into different models (cf. Section 4). Each model addresses different kinds of changes.

We evaluate the utility of our approach by showing how a number of change scenarios are supported by the RuleCaster system and the underlying programming model. This method is inspired by work on scenario-based analysis of software architectures [14, 13]. Scenarios allow us to express particular instances of a quality attribute important to specific life-cycles of an application.

The change scenarios are based on the scenario introduced in Section 3.2. We can assume that the infrastructure is installed and the initial application deployed. This allows us to focus on the more interesting change scenarios. Table 1 shows six probable changes that will occur during the lifetime of our smart home example and its applications.

In the following we discuss how the concept of programmable space and RuleCaster support each individual scenario. We address each scenario from the perspective of the application developer who has to make the changes:

Scenario 1. In contrast to the existing scenarios it would be difficult to assign this scenario to an existing space; it concerns the whole house. Hence, we implement the application logic for the space `house`. A node can belong to several spaces and in our infrastructure every node is statically assigned to the space `house`. Figure 10 shows the code we add to the application definition. There are two alternative conditions that define the transition to a state `lightsOff` (i.e. the lights are turned off). The first condition is satisfied if there is no observed motion (`motion(X), X=0`) and the lights are then automatically turned off (`light(0)`). The second condition is satisfied if a person is in bed (`bed(X), X>20`) and the lights are then automatically turned off (`light(0)`). After these changes the code is recompiled and automatically re-deployed in the network. Because the smart home is represented as programmable space, we only have to make changes to the high-level application code (i.e. the logical structure) of the application.

Table 1: Change scenarios.

	Change scenario	Example	Concern
1.	Extend application.	Alice is an energy-conscious person. She wants to have the smart home modified so that she can save energy by having the lights turned off when nobody is in the house or she is in bed.	Logical structure
2.	Change the distribution of the application.	Some nodes have been connected to a permanent power supply. They should therefore contribute more computational power to the execution of the application.	Physical structure
3.	Add new nodes to the infrastructure.	The motion sensor in the corridor does not cover the whole area of the L-shaped room. Hence, a new motion sensor is added to cover the other part of the corridor.	Infrastructure
4.	Extend infrastructure.	Alice has read about the danger of carbon monoxide. She decides to have a carbon monoxide detector installed in the kitchen.	Infrastructure
5.	Extend application.	Alice wants all windows to automatically open if a high concentration of carbon monoxide is measured. Furthermore, she wants the alarm to be actuated immediately.	Logical structure
6.	Move application.	Her parents like her smart home and decide to install a computer infrastructure in their house. They want to run the same applications as the ones in their daughter's smart home.	Physical structure

```
1     SPACE(house) {
2       INTERFACE:
3         SENSOR(motion/1),
4         ACTUATOR(light/1).
5
6       PRE_STATE() [
7         STATE :- motion(X), X=0, light(0).
8         STATE :- bed(X), X>20, light(0).
9       ] POST_STATES(lightsOff).
10
11      ...
12      ...
13    }
```

Figure 10: Definition of the space `house`.

Scenario 2. We do not have to change the application code. The network model reflects the infrastructure changes to the compiler, which can find a better task assignment to the nodes. In the changed application the nodes connected to a permanent power supply take over some tasks previously executed by other nodes. This can have a positive effect on the energy consumption of the battery-powered nodes. The application is recompiled and automatically re-deployed. The changes only affect the physical structure of the application.

Scenario 3. After having added the new node configured with the RuleCaster middleware and assigned to the spaces `corridor` and `house`, we recompile the application and re-deploy it. We do not have to change the application logic because the changes do not affect the logical structure of the application. The new motion sensor is used in the same way as the existing sensor in the corridor.

Scenario 4. The carbon monoxide detector is installed in the kitchen and configured with the RuleCaster middleware, which includes the assignment to the relevant spaces (i.e. `kitchen` and `house`). Because the current smart home application does not require this sensor, it does not have a direct effect on the running application. The application logic is extended to use this sensor in the next scenario.

Scenario 5. This scenario only affects the application code of the space `house`. We add a new state `highCOConcentration` that describes the situation of high carbon monoxide concentration. We also add the required conditions for a transition into this state (i.e. observed high carbon monoxide concentration). This state can then be used to define the condition for opening the windows. This change scenario only affects the application code. After recompilation the application is automatically re-deployed and then executed.

Scenario 6. Her parents' computer infrastructure is set up differently. Due to their bigger house, they have installed more motion sensors. Because the application logic stays the same, the application developer only has to recompile it to generate the adapted physical structure of the application before it is re-deployed.
These scenarios show that specific changes can be attributed to separate models. This separation reduces the complexity of changes. Changes to any of these models can be directly propagated to the running application by recompilation and re-deployment. Hence, RuleCaster does not only address the implementation of smart home applications but also supports evolutionary changes after the initial deployment in a unified way.

8 Conclusion

Our language-based approach provides support for programming and maintaining the smart home through a number of measures:

- Our approach does not force the application developer to express the global behaviour of an application in terms of local actions taken at individual nodes. This method would be cumbersome because the pro-

grammer has to deal with many issues related to distributed programming which make application development difficult, time-consuming and error-prone. Instead, the application developer can program the smart home as one logical entity.

- The high-level language eases the implementation efforts for the application developer. People are inclined to use rules for describing the required behaviour of a smart space. By providing a rule-based language we can decrease the mental gap between the user-based application description and the actual implementation.

- By separating application development into different high-level models we can simplify application maintenance due to changing user requirements or changes in the living environment. We show that different change scenarios in the life-cycle of an application address distinct models.

To successfully realise the vision of a smart home outside research labs, we have to address issues related to programmability and maintenance of the related computing infrastructure. We believe that the proposed system is a step into the right direction.

References

[1] Zigbee Alliance. Zigbee specification. http://www.zigbee.org, December 2006.

[2] Juan C. Augusto and Chris D. Nugent. Smart homes can be smarter. In *Designing Smart Homes - The Role of Artificial Intelligence*, 2006.

[3] Urs Bischoff and Gerd Kortuem. Programming the ubiquitous network: A top-down approach. In *Proceedings of the System Support for Ubiquitous Computing Workshop (UbiSys'06)*, 2006.

[4] Urs Bischoff and Gerd Kortuem. Rulecaster: A macroprogramming system for sensor networks. OOPSLA Workshop on Building Software for Sensor Networks, 2006.

[5] Urs Bischoff and Gerd Kortuem. Rulecaster: A programming system for wireless sensor networks. In *Proceedings of the First European Conference on Smart Sensing and Context (EuroSSC'06)*, 2006.

[6] Anind D. Dey, Timothy Sohn, Sara Streng, and Justin Kodama. iCAP: Interactive prototyping of context-aware applications. In *Proceedings of the 4th International Conference on Pervasive Computing (PERVASIVE 2006)*, 2006.

[7] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2):97–166, 2001.

[8] W. Keith Edwards and Rebecca E. Grinter. At home with ubiquitous computing: Seven challenges. In *UbiComp '01: Proceedings of the 3rd international conference on Ubiquitous Computing*, pages 256–272, London, UK, 2001. Springer-Verlag.

[9] Mareca Hatler and Chalie Chi. Wireless sensor networks: Growing markets, accelerating demand. On-World, July 2005.

[10] Sumi Helal. Programming pervasive spaces. *IEEE Pervasive Computing*, 4(1):84–87, 2005.

[11] Sumi Helal, William Mann, Hicham El-Zabadani, Jeffrey King, Youssef Kaddoura, and Erwin Jansen. The gator tech smart house: A programmable pervasive space. *Computer*, 38(3):50–60, 2005.

[12] Stephen S. Intille. Designing a home of the future. *IEEE Pervasive Computing*, 1(2):76–82, 2002.

[13] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements. Scenario-based analysis of software architecture. *IEEE Software*, 13(6):47–55, November 1996.

[14] Rick Kazman, Leonard J. Bass, Mike Webb, and Gregory D. Abowd. SAAM: A method for analyzing the properties of software architectures. In *International Conference on Software Engineering*, pages 81–90, 1994.

[15] Cory D Kidd, Robert Orr, Gregory D. Abowd, Christopher G. Atkeson, Irfan A. Essa, Blair MacIntyere, Elizabeth Mynatt, Thad E. Starner, and Wendy Newstetter. The aware home: A living laboratory for ubiquitous computing research. In *Proceedings of the International Workshop on Cooperative Buildings (CoBuild 1999)*, pages 191–198, 1999.

[16] Tom Rodden and Steve Benford. The evolution of buildings and implications for the design of ubiquitous domestic environments. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 9–16, New York, NY, USA, 2003. ACM Press.