A Compiler for the Smart Space

Urs Bischoff, Gerd Kortuem

Lancaster University, Lancaster LA1 4YW, UK

Abstract. Developing applications for smart spaces is a challenging task. Most programming systems narrowly focus on the embedded computer infrastructure and neglect the spatial aspect of this fusion between a physical and a virtual environment. Hence, application logic is not implemented for the smart space but for the embedded network, which is only one aspect of the system. Our programming system supports an abstract model of a smart space. A high-level language is used to implement the application logic for this model. In this paper we show how a compiler translates code written for this abstract model into a distributed application that can be executed by a computer infrastructure. The compiler allows for a clear separation between the application code and its execution in a concrete network. This simplifies the development and maintenance of an application because the application programmer can focus on the actual application logic for the smart space instead of issues related to a concrete network.

1 Introduction

High-level programming languages provided the basis for the development of complex computing systems. Compared to assembly programming early programming languages (e.g. COBOL or FORTRAN) simplified the programming task considerably. Even though these languages were developed in the 50's, millions of lines of code written in these languages are still in use today. The secret behind this success is that they were designed for an abstract computer model instead of a concrete hardware platform. A compiler is responsible for the translation between the code written for the abstract computer model and code that can be executed by the computer hardware. This separation allowed the independent development of hardware and software over the years.

Today, we are working on the ubiquitous computing vision of smart spaces consisting of large numbers of sensors and actuators. And we are faced with the challenging task of programming this new type of computer infrastructure. Similar to how assembly programming did not scale to complex applications, system design focused on individual network nodes does not scale to networks consisting of large numbers of sensor and actuator nodes. We must consider how to model these smart spaces and design suitable programming abstractions.

A smart space embeds interconnected technologies that are, more or less, responsive to the state of the environment. A problem pointed out in [1] is that our experience in building smart spaces 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 infrastructure of a smart space every few months. As emphasised by Rodden and Benford [2], 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 of a smart space 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 a novel programming paradigm and compiler for smart spaces. In our programming paradigm a smart space is not just the end product of a fusion between a physical and virtual environment but it is a fundamental abstraction that is used during the whole life cycle from the design to the deployment of such a space. The application developer does not implement the application logic for a network that happens to be embedded in a physical environment, but for an abstraction of the actual space that should exhibit smart behaviour. A high-level language is used to define the application logic for this space abstraction. A compiler translates the high-level application code into a representation that can be executed by the embedded network. It splits the application code into a set of tasks and operators, and assigns them to individual network nodes. This top-down approach allows the application programmer to move away from system problems of the individual nodes to the actual programming problem of the smart space as a whole. Hence, the global application does not have to be expressed in terms of a set of distributed tasks for individual nodes. Furthermore, because the logic is implemented for the smart space itself (and not for the embedded network), the network infrastructure can evolve independently of the application logic.

This paper is organised as follows. Section 2 presents the abstract concept of a programmable smart space. In Sec. 3 we describe our programming system that integrates this smart space model. Section 4 introduces an application scenario that is used in this paper to illustrate the technical detail of the compiler addressed in Sec. 5. In Sec. 6 we focus on the mapper, which is a core component of the compiler. We evaluate it in terms of a case study. Section 7 gives a brief overview of related work. Finally, Sec. 8 concludes this paper.

2 Programming a Smart Space

An application programmer wants to write an application for a space to make it "smart". For example, a room should turn off its lights if nobody is inside. This point of view is in contrast to the prevalent programming paradigm focused on the embedded network infrastructure. Implementing this example, the application programmer would have to address the motion sensors to find out whether there is motion in the room. And if no motion is detected, a command can be sent to the light switch to turn the lights off. The ubiquitous computing vision of technologies that *weave themselves into the fabric of everyday life until they are indistinguishable from it* [3] cannot support such an explicit technology focus on programming. Our novel programming paradigm supports the definition of the application logic for the smart space. The actual technology disappears in the background; it is irrelevant for the logic and is interchangeable.

Figure 1 illustrates this programming paradigm. It shows a floor of an office building. Each room should be made "smart" by embedding some logic into it. For example, each room should turn off its light when nobody is inside. Additionally, the office occupants might want to individualise their office and update it with additional logic. The kitchen is a different type of space which requires different logic; it should detect when the stove is turned on and nobody is monitoring it. Finally, there is a fire alarm that concerns the whole floor. What is important to notice is that we want to embed logic into the actual room. The embedded sensor and actuator network is not relevant at this point. Our programming paradigm supports this point of view. By focusing on the space instead of the embedded computer infrastructure, we can separate the implementation of the application logic from the underlying hardware. The application developer can deal with the space abstraction instead of the details of the underlying network such as types and number of nodes. A compiler is responsible for mapping the application logic onto the computer infrastructure of a smart space.

2.1 Abstract Smart Space Model

In order to support this programming paradigm, an abstract model of a smart space is necessary. Figure 1 shows that there can be several separate smart spaces (e.g. offices and kitchen), which are all contained within another smart space (office floor). And each space exhibits its own logic. Communication between these spaces might be necessary (e.g. the kitchen might want to send an alarm to an occupied office if the stove is turned on).

Figure 2 illustrates an example of the abstract model of smart spaces. There are several smart spaces. Some are contained within another one. Each smart space has four distinct interfaces. One interface is used to observe the environment and another one to affect the environment. The two remaining interfaces provide the communication channels to send/receive state information to/from other smart spaces. A space uses its sensor observations to update its internal state. It can then use this state information to affect the state of the physical environment.

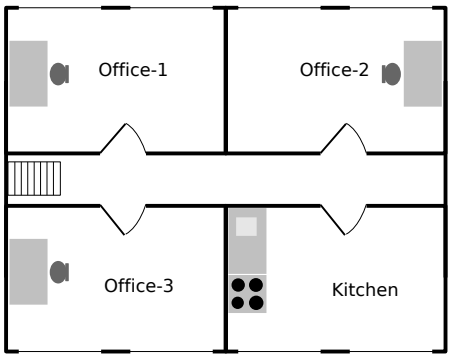

Fig. 1. A floor in an office building as a smart space.

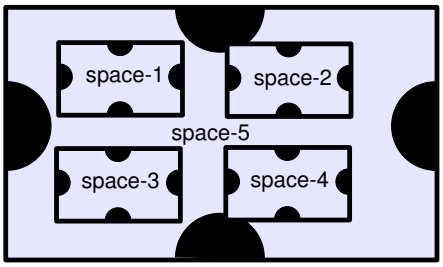

Fig. 2. Example of the abstract model of five smart spaces. Each one has interfaces for communication with other spaces and the physical environment.

2.2 Infrastructure Assumption

The computer model of a standard high-level programming language is based upon several requirements for the the underlying computer (e.g. computer has memory and can do comparisons). Similarly, the abstract smart space model makes some assumptions about the structure and capabilities of the underlying network:

- Nodes in the network can communicate with each other.
- A node belongs to a symbolic location that it is assigned to or it can assign itself to (e.g. node 23 is in room 213). This information is necessary to find a mapping between the smart space model and the actual network infrastructure.
- Properties of network nodes are known (e.g. node 23 has a temperature sensor). The nodes could either describe themselves or the properties are well-known facts.

3 System Overview

The programming paradigm introduced in Sec. 2 is directly supported by our system called RuleCaster. It provides a high-level language and implementation for the smart space model. In the following we outline this system with a focus on the two main problems mentioned earlier: (1) defining the global application in terms of separate local actions, and (2) accommodating changes.

By dealing with smart space not only as a physical entity but also as an abstract computer model we realised the concept of a programmable space. The advantage of this approach is that the application developer can implement applications for the smart space as one logical entity; *the smart space is the*

computer. The application developer is provided 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 space application undergoes throughout its life-cycle. 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 network 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, the infrastructure might need to be updated when a new generation of hardware devices becomes available with different processor, memory or radio. Another example is modifying the infrastructure by adding or removing nodes.

Our programming system supports these classes of changes by separating them into three separate models: logical structure, physical structure and infrastructure. Changes to any of these three models can be directly propagated to the running application by recompilation and re-deployment.

Figure 3 illustrates the architecture of our system. The logical structure of an application is defined in the RuleCaster Application Language (RCAL) — a high-level language designed for the smart space model. The physical structure is implicitly described by the compiler that translates the logical structure into an executable representation.

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).

In this paper we exclusively focus on the description of the compiler and the generation of the physical structure of an application.

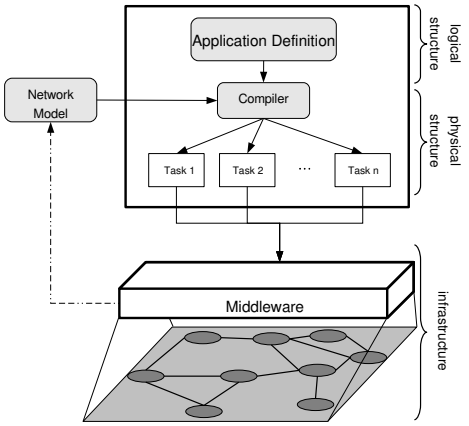

Fig. 3. The system architecture.

4 Application Scenario

In order to simplify the discussion in the remainder of this paper we introduce a simple application scenario. As an example we use a smart office building.

4.1 Scenario

Bob is the environmental officer at Green University. He wants to reduce the electrical energy consumption. One of the first ideas is to turn off the lights in an office if there is enough day light coming through the windows or if nobody is in the office. And instead of using the HVAC system to control the temperature, he wants the windows to be opened automatically if the outside temperature can be used to reduce the office temperature.

4.2 Scenario Implementation in RCAL

The scenario application is implemented in RCAL — a state-based programming language for a smart space (i.e. office). Based on sensor observations the smart space determines the current state of the space. Rules are used to define the state transitions. And states can trigger the activation of an actuator. More information about this language can be found in [4]. Figure 4 shows a simplified version of the application code. For space reasons we only show a small part of the whole application, which we are going to use in the rest of this paper. Communication with other spaces is not shown. Lines 2-8 declare the interface of the office space. For example, line 3 declares a sensor `indoorTemperature` that delivers one value at a time (i.e. the indoor temperature).

Lines 10-14 define the first state transition rule. This rule is applied if the space is in a state `lightOn` (the rule that defines this state is not shown). If the

rule in line 11 or the one in line 13 is satisfied, the state `lightOn` changes into a state `lightShouldBeOff`, which means that the lights should be turned off. The rule in line 11 is satisfied if both conditions `light.average(X)` and `bright(X)` are satisfied. The condition `light.average(X)` delivers the average light sensor reading of this space in the variable `X`. The condition `bright(X)` is defined in a rule in line 12. It accepts the light value `X` as input. It is satisfied if this value is more than 20. Similarly the rule in line 13 is satisfied if the observed motion is less than 2.

Lines 16-18 define the transition from the state `lightShouldBeOff` to the state `lightOff`. This transition depends on the rule in line 17 that is satisfied if the condition `lightSwitch(0)` is satisfied. As a side-effect this condition turns the light off.

```
1    SPACE(office) {
2    INTERFACE:
3      SENSOR(indoorTemperature/1),
4      SENSOR(outdoorTemperature/1),
5      SENSOR(motion/1),
6      SENSOR(light/1),
7      ACTUATOR(window/1),
8      ACTUATOR(lightSwitch/1).
9
10   PRE_STATE(lightOn) [
11     STATE :- light.average(X), bright(X).
12     bright(X) :- X>20.
13     STATE :- motion(X), X<2.
14   ] POST_STATES(lightShouldBeOff).
15
16   PRE_STATE(lightShouldBeOff) [
17     STATE :- lightSwitch(0).
18   ] POST_STATES(lightOff).
19
20   ...
21   ...
22
23   }
```

Fig. 4. The implementation of the application scenario.

4.3 The Smart Space Model and Infrastructure

In order to execute the application logic a smart space requires certain services from the underlying infrastructure. Figure 5 depicts the smart space model of an office space on the left hand side and the infrastructure of an office on the right hand side. The logic implemented for the smart space model requires a number of services for observing and affecting the physical environment: for example, the service `indoorTemperature/1` is required to get the indoor temperature. The infrastructure can provide these services with its sensors and actuators. The compiler is responsible for mapping the smart space model onto a concrete infrastructure. In general there is not a one-to-one mapping; as shown in this example, the indoor temperature is measured by three different sensor nodes.

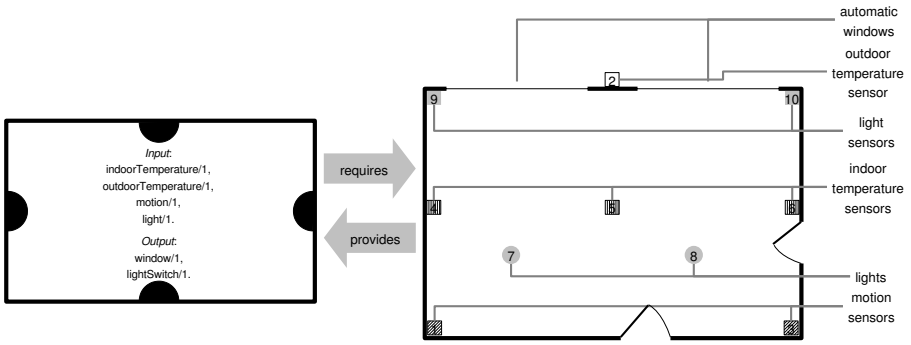

Fig. 5. The smart space model and a possible infrastructure.

5 Compiler

Applications are written for an abstract computer model of a smart space. A compiler has to find a mapping between this abstract model and the concrete smart space infrastructure. In other words it has to translate the high-level application code into executable code for the network. The compiler consists of several subcomponents that implement the different phases of the whole compilation process. Figure 6 depicts these components. In the following we address each component individually.

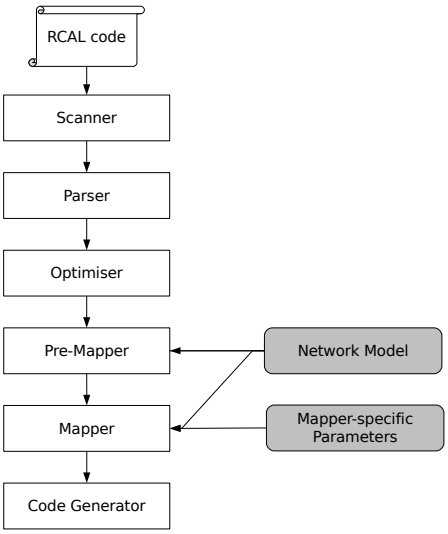

Fig. 6. The architecture of the compiler.

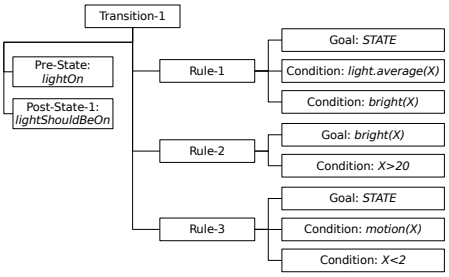

Fig. 7. The syntax tree of a state transition rule.

5.1 Scanner

The scanner reads the application code and builds an internal representation of the syntax as a tree. State transitions are the core part of the application code. In the following we focus on one transition to illustrate the different compiler phases. Figure 7 depicts the syntax tree of the transition rule in lines 10-14 of Figure 4. This tree is a direct translation of the application code syntax. The root node is the transition node, which has a pre-state, one post-state and three rule nodes as children. Each rule node has a goal and several condition children. Additionally there is a number of other nodes in this tree that represent the other elements of the application code, which are not shown.

5.2 Parser

The input to the parser component is the syntax tree. The parser translates this syntax representation into a semantic representation. A semantic representation contains information about the actual execution of the application. The core part of the semantic representation are the task graphs. Each transition rule tree is transformed into a task graph consisting of several operators connected with communication channels. The parser proceeds in several steps:

1. Initialise a *START*, *OR* and *END* operator and connect them with a channel.
2. Extract the top-level state rules. These are the rules whose satisfaction cause a state change. Their rule goal is `STATE`. Initialise a representative operator for each condition and connect them with a channel. Rename the variables because the scope of a variable is changed from the individual rule to the whole task graph. Attach these operators to task graph between the *START* and *OR* operators.
3. Traverse task graph and replace operators whose corresponding condition is defined by an additional rule (e.g. the condition `bright(X)` is defined by a rule). The rule is transformed into an operator-channel list and integrated into the task graph at the replaced operator's position. The variables are renamed to conform with the existing variable names.
4. Assign types to operators. Operators that correspond to a sensor or actuator interface declaration are labelled with either *SENSOR* or *ACTUATOR*.

Figure 8 depicts the task graph generated by the parser. Each operator represents a condition. A task graph specifies the execution of a transition rule. Execution starts at the operator labelled with *START* and proceeds along the outgoing channels towards the operator *END*. In this task graph there are two parallel execution paths. The execution along an execution path is stopped if the evaluation of a condition fails (e.g. if the motion value *V2* is less than 2). If one execution path reaches the *END* operator, it causes a transition from the pre-state to the post-states.

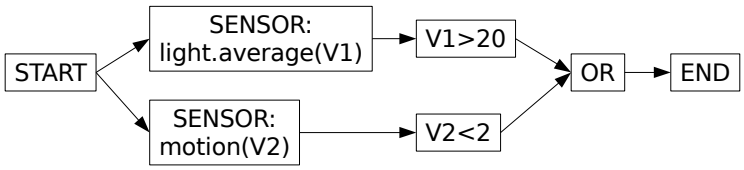

Fig. 8. The task graph of a rule.

5.3 Optimiser

The optimiser is an optional component. It accepts a task graph as input and produces a task graph as output. The goal of the optimiser is to minimise the flow of information between the operators by rearranging them without altering the observable behaviour of the application. Actuators need special attention because they provide the interface for output to the environment, which is directly observable.

The optimiser of our system tries to "push" an operator that uses data as close as possible to the operator that produces the relevant data. The following operation is applied to rearrange operators:

> Two operators that are directly connected with a channel can swap position if both have exactly one input and one output channel, none of them is an *ACTUATOR*, *START* or *END* operator and the successor operator does not depend on data of the predecessor operator.

The optimiser cannot rearrange any operator in our example because neighbouring operators depend on each other; e.g. the operator with condition *V2<2* depends on the output of *motion(V2)*. Figure 9 shows a different example where two operators could be swapped. The advantage of the new task graph is that the execution could already be stopped at the second operator if the comparison condition cannot be satisfied.

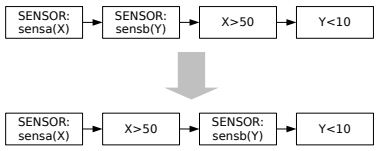

Fig. 9. The optimiser rearranges the operators of the task graph.

5.4 Pre-Mapper

The pre-mapper component prepares the task graph for the actual mapping process. As shown in Figure 6 it requires the network model as input. The

network model contains an abstract representation of every network node. This representation consists of three parts:

Services. Each node offers a number of services. For example, a node that has a light sensor, offers a sensor service that can deliver a light value. Or another node provides a service that compares two numbers.

Attributes. Each node has some static features called attributes. The attributes that are used by the pre-mapper are the symbolic locations. Each node can belong to several locations (e.g. node-3 belongs to *office* and *building-CS*).

Properties. Properties are dynamic features of nodes. The battery level or connectivity information are properties. The pre-mapper does not need any node property information.

The pre-mapper is responsible for the following steps:

1. It identifies each sensor operator in the given task graph. If the operator explicitly specifies an aggregator (e.g. average), it splits the operator into an aggregator operator and a sensor operator. Otherwise it attaches a generic OR aggregator to the sensor operator.
2. For each sensor/actuator operator in the task graph it counts all nodes in the respective space that provide the corresponding sensor/actuator service. Then, it splits the corresponding sensor/actuator operator into several instances and assigns one to each of the corresponding sensor/actuator service nodes (i.e. it labels the operator with the id of the node).

The output of the pre-mapper component is depicted in Figure 10. After the pre-mapper phase every sensor and actuator operator is assigned to a network node.

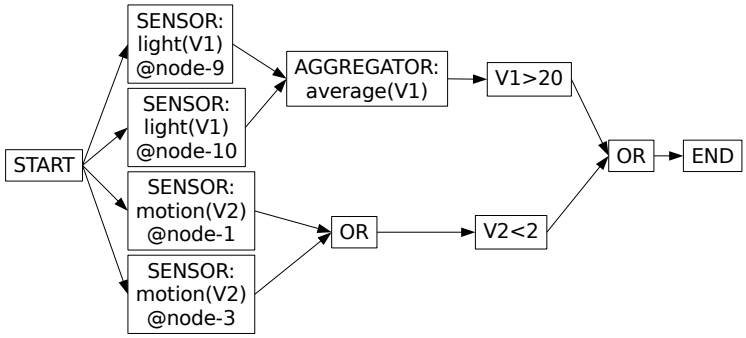

Fig. 10. The pre-mapper splits the sensor/actuator operators into several instances and assigns each one to a separate node that can provide the required service.

5.5 Mapper

The mapper is the core component of the compiler. It assigns the operators (that have not been assigned yet by the pre-mapper) given by the task graph to nodes in the smart space network. Those nodes will be responsible for the execution of the assigned operators. Similarly to the pre-mapper it requires network information given by the network model.

The observable behaviour of an application is defined in the logical structure of the application. The observable behaviour must not be changed by the mapper. However, the mapper can influence the quality of an application. For example, a centralised solution and fully distributed solution have different characteristics in terms of reliability and energy consumption [5]. Depending on the requirements the compiler can use a different mapper. And different mapper components require different mapper-specific parameters.

In general terms, the mapping problem can be reduced to finding a matrix P such that $P(k,i) = 1$ if operator k is assigned to node i under the constraint C. We assume that an operator k can only be assigned to one node i, thus $P(k,j) = 0$ for all $j \neq i$. Although redundant placement of operators can be useful, we do not consider this in the paper. Let C denote a constraint matrix such that $C(k,i) = 0$ if operator k has to be assigned to node i, $0 < C(k,i) < \delta$ if operator k can be assigned to node i, and $C(k,i) = \delta$ if operator k cannot be assigned to node i given a constant δ. The elements of the constraint matrix could be seen as preferences to where operators should be placed. Or they could be interpreted as the cost for executing a certain operator k on a node i. We will give more details about two specific mapper components in Sec. 6.

The output of the mapper component is a task graph whose operators are assigned to network nodes. Figure 11 depicts the assignment of a mapper.

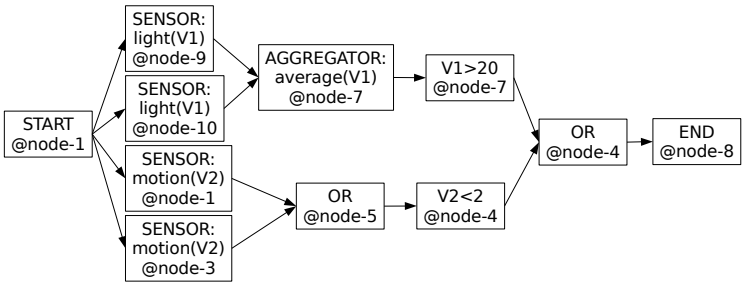

Fig. 11. The mapper assigns every operator of the task graph to a network node.

5.6 Code Generator

The code generator transforms the task graph into byte code that can be executed by the stack machine interpreter in our system-specific middleware. The

execution of an application is based on the idea of a token that is passed from operator to operator along the execution paths. Tokens transport relevant sensor data from one operator to the next one. Each node has a handler that accepts tokens and forwards them to the required operator. An operator extracts the relevant data from the token and evaluates the given rule condition. If the condition cannot be satisfied, the token is killed. Aggregators combine several input tokens into one output token. For each output channel the operator generates a token containing the data required by its successor operators and sends it to the node executing the first successor operator.

Figure 12 illustrates the byte code of an operator that is executed by *node-7*, which is responsible for the evaluation of the condition *V1>20*. Operators send and receive data through channels that connect them with other operators. In line 1 data field 0 of the token in incoming channel 0 is loaded and pushed onto the stack. The value 20 is pushed onto the stack in line 2. The command `eval` calls a node service; in this example it is the service `lessThan`, which pops the two top numbers from the stack and compares them. If the comparison succeeds, it pushes the value 1 onto the stack. If it fails, the value 0 is pushed onto the stack. In line 4 the top stack value is removed and compared to 0. If it is equal to 0, execution jumps over the next line. If the top stack value is 1, execution continues at line 5, where the send command tells the outgoing channel 0 to send a token through the channel to the next operator. These channels are set up at the beginning when the operators are deployed. In this example, the token does not have to carry any application data (e.g. sensor data) because successor operators do not require any specific data. The `end` command in line 6 finishes execution.

```
1    iload 0 0 //push data field 0 from channel 0 onto stack
2    ipush 20 //push the value 20 onto the stack
3    eval lessThan //call the service lessThan
4    ifeq 2//jump over next line if service evaluation has returned 0
5    send 0 //send data in channel 0
6    end //operator execution finished
```

Fig. 12. Operator code that is interpreted by the middleware of our system.

6 Mapper Component

The mapper component of the compiler was introduced in Sec. 5.5. It is the core part of the compiler. Deciding where operators are executed and states are stored in the network are its main tasks. The decision where the states are stored is equivalent to the decision where the START and END operators of a task graph are placed.

In this section we describe two exemplary mappers that can be selected as compiler plugins at compile-time. Both mappers assign task operators to nodes

that are contained in the smart space that is described by the smart space model; i.e. if the smart space is an office, all operators are executed by nodes that are actually in the office. In general this is not a necessary requirement, because *free* operators, which are operators that are not of type *SENSOR* or *ACTUATOR*, could be executed anywhere; e.g. sensor data could be analysed anywhere.

Both mappers initialise the following parameters for mapping the task t (cf. Sec. 5.5):

- S is the set of nodes contained in the space of t.
- $C(k, i) = 0$ if operator k has already been assigned to node i by the pre-mapper.
- $C(k, i) = 1$ if node $i \in S$ and node i can execute operator k.
- $C(k, i) = 2$ if node $i \notin S$ or node i cannot execute operator k.
- $P(k, i) = 0$ for every k and i.

6.1 Centralised Operator Mapping

The first mapper chooses one random node in the space that corresponds to the task that has to be mapped onto the infrastructure. It then assigns every free operator to this node. If it cannot find such a node that is able to execute the required operators, the compilation fails and the mapper returns a description of the additional requirements for the infrastructure. Pseudocode of the mapper is shown in Figure 13.

```
1     for each k do
2       if there is i such that C(k,i)=0 then
3         P(k,i) = 1; mark k;
4
5     find random i such that
6       for each unmarked k C(k,i)==1
7     if found i then
8       for every unmarked k
9         P(k,i) = 1;
10    if not found i then
11      ERROR;
```

Fig. 13. Algorithm of centralised operator mapping.

Figure 15 illustrates the output of this mapper when applied to our example task. The sensor operator on nodes 1, 3, 9 and 10 send their data to node 7, which executes the remaining operators and stores the state information.

6.2 Decentralised Operator Mapping

In contrast to the first mapper, the decentralised operator mapper chooses a different random node for each free operator. Similar to the first mapper this mapper can also fail to find a node for every operator. In such a case it can

```
1     for each k do
2       if there is i such that C(k,i)=0 then
3         P(k,i) = 1; mark k;
4
5     for each unmarked k do
6       find random i such that
7         C(k,i)==1
8       if found i then
9         P(k,i)=1; mark k;
10      if not found i then
11        ERROR;
```

Fig. 14. Algorithm of decentralised operator mapping.

provide an exact description of the additional requirements for the infrastructure. Figure 14 illustrates algorithm of the mapper component.

Figure 16 depicts the output of this mapper. The sensor operators on nodes 9 and 10 send their tokens to an operator on node 7 that computes the average. This average is compared with 20 by another operator on node 7. Then, the token is forwarded to node 4. Sensor nodes 1 and 2 send their tokens to an aggregator on node 5. The resulting token is forwarded to node 4. Node 4 sends the token to the *END* operator on node 8.

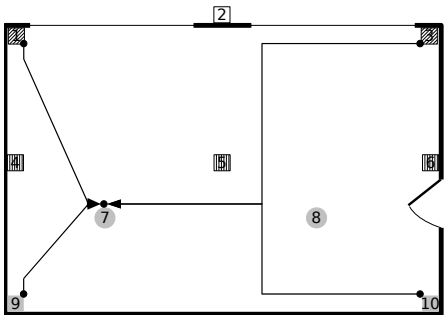

Fig. 15. The first mapper assigns operators in their corresponding space centrally.

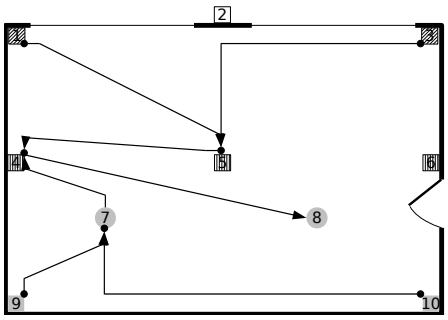

Fig. 16. The second mapper assigns operators to random nodes in their corresponding space.

6.3 Discussion

The observable behaviour of the application is not directly affected by the chosen mapper. However, the mapper can influence the qualitative (or non-functional) aspects such as energy consumption, privacy or reliability of the running application. The main focus of this paper is on the technical aspects of the compiler; and we want to leave a qualitative discussion open for future work.

7 Related Work

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 space, but in general) shows great potential beyond research labs (e.g. [6]). 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 [7]).

This technological development and the trend of moving away from a single computer per user has cleared the way to the development of smart spaces. Several research groups have developed first-generation prototypes of the smart home as an example of a smart space to study the computing needs in our everyday lives ([8–10]).

Currently, the main technological focus is on integrating suitable hardware into the environment, 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.

RuleCaster is related to *macroprogramming* approaches, which have emerged as a potential solution for simplifying the development of applications for these embedded distributed computers. Common to all these approaches is that they address the whole network as one programmable unit instead of the individual nodes.

The Kairos macroprogramming system [11] extends traditional programming languages with three specific abstractions: node, one-hope neighbours of a node and remote data access. Its runtime system supports the execution of applications using these abstractions. In contrast to Kairos RuleCaster provides node-independent abstractions. A RuleCaster application is expressed in a network-independent way. Kairos provides node-dependent abstractions and therefore forces the programmer to express global application behaviour in terms of nodes and node state. Accordingly, every node executes the same code. The RuleCaster compiler has more freedom and generates specific code for each individual node depending on the properties and capabilities of the respective node, which is more suitable for a heterogeneous network environment.

Similar to RuleCaster Regiment [12] is also based on a declarative language. Functional programming constructs are used to build applications. As a fundamental programming abstraction it uses the concept of a region, which is a collection of data signals originating from a set of nodes. Regiment applications describe the manipulation of regions. This is related to the abstract smart space model used in RuleCaster. However, Regiment addresses a different class of applications. While RuleCaster focuses on state-based sensor-actuator applications, Regiment is designed for extracting time-varying sensor data streams from a sensor network.

COSMOS [13] is comprised of a programming language and an operating system. Similar to Regiment it addresses processing and aggregation of time-varying sensor data streams. The basic building blocks of a COSMOS application are functional components (FCs) that transform input streams into output streams. These FCs are interconnected via asynchronous data channels to implement a sensor network application. Compared to RuleCaster, COSMOS is on a much lower abstraction level which makes application development hard and error-prone. The assignment of functional components to nodes (or classes of nodes) is specified by the application programmer. This simplifies the mapping process but makes the development of complex applications difficult.

At the core of an ATaG [14] macroprogram are abstract tasks. An abstract task encapsulates the processing of data. The flow of information between abstract tasks is defined in terms of input/output relations. To achieve this, abstract channels are used to connect tasks. ATaG is a suitable approach for implementing high-level sensor network applications. However, the separation between the logical structure and the physical structure is not as strict as in RuleCaster. On the one hand this simplifies the task of the compiler. On the other hand, it makes it more difficult to change the physical structure of an application.

8 Conclusion

Our compiler-based approach provides support for programming and maintaining a smart space through a number of measures:

- Our approach does not force the application programmer to express the application logic of a smart space in terms of distributed actions for the underlying infrastructure. This method would be cumbersome because the programmer has to deal with many issues related to distributed programming which make application development difficult, time-consuming and error-prone. Instead the application programmer can program the smart space as one logical entity. And the compiler is responsible for the distribution.
- Our smart space model addresses the evolutionary changes of a smart space by separating the infrastructure from the application logic. They can evolve independently. The compiler accommodates changes by finding a new mapping of the application logic onto the infrastructure.

Smart spaces promise a visionary concept about how computers and the physical environment can create an integrated system. If we want to realise this vision outside research labs, we have to address issues related to programming and maintenance of the related computer infrastructure. We believe that providing a unified abstraction of the environment and the embedded infrastructure in terms of a smart space model is a step into the right direction.

References

1. Helal, S.: Programming pervasive spaces. IEEE Pervasive Computing **4**(1) (2005) 84–87

2. Rodden, T., Benford, S.: The evolution of buildings and implications for the design of ubiquitous domestic environments. In: Proceedings of the SIGCHI conference on Human factors in computing systems, New York, NY, USA, ACM Press (2003) 9–16
3. Weiser, M.: The computer for the 21st century. Scientific American (1991)
4. Bischoff, U., Sundramoorthy, V., Kortuem, G.: Programming the smart home. In: Proceedings of the 3rd IET International Conference on Intelligent Environments. (2007)
5. Ahn, S., Kim, D.: Proactive context-aware sensor networks. In: Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN). (2006)
6. Hatler, M., Chi, C.: Wireless sensor networks: Growing markets, accelerating demand. OnWorld (2005)
7. Alliance, Z.: Zigbee specification. http://www.zigbee.org (2006)
8. Kidd, C.D., Orr, R., Abowd, G.D., Atkeson, C.G., Essa, I.A., MacIntyere, B., Mynatt, E., Starner, T.E., Newstetter, W.: The aware home: A living laboratory for ubiquitous computing research. In: Proceedings of the International Workshop on Cooperative Buildings (CoBuild 1999). (1999) 191–198
9. Intille, S.S.: Designing a home of the future. IEEE Pervasive Computing **1**(2) (2002) 76–82
10. Helal, S., Mann, W., El-Zabadani, H., King, J., Kaddoura, Y., Jansen, E.: The gator tech smart house: A programmable pervasive space. Computer **38**(3) (2005) 50–60
11. Gummadi, R., Gnawali, O., Govindan, R.: Macro-programming wireless sensor networks using Kairos. In: Distributed Computing in Sensor Systems: First IEEE International Conference (DCOSS 2005). (2005)
12. Newton, R., Morrisett, G., Welsh, M.: The Regiment macroprogramming system. In: IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks, New York, NY, USA, ACM Press (2007) 489–498
13. Awan, A., Jagannathan, S., Grama, A.: Macroprogramming heterogeneous sensor networks using COSMOS. SIGOPS Oper. Syst. Rev. **41**(3) (2007) 159–172
14. Pathak, A., Mottola, L., Bakshi, A., Prasanna, V.K., Picco, G.P.: Expressing sensor network interaction patterns using data-driven macroprogramming. In: Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications Workshops. (2007)