Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

KAFCA: Knowledge Autonomy for Reactive Context-aware Applications

Neil O'Connor

A thesis submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

July 2010

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work. I agree that Trinity College Library may lend or copy this thesis upon request.

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Neil O'Connor

Neil O'Connor

Dated: 4th June 2010

Acknowledgements

I would firstly like to sincerely thank my supervisor Prof. Vinny Cahill for his help and guidance throughout this process. It has been an invaluable learning experience.

I have been fortunate enough to be surrounded by interesting, friendly and supportive people in the Distributed Systems Group, and you have made this a fantastic place to work. In spite of my self-perceived hardship I think I will look back on these years as some of my best, and I hope that some of you will remain my friends for life.

I would also like to thank my family, who have never complained that I'm taking too long. Thank you for never questioning that I'd get there in the end.

Finally to Jen, who has been there through every high and low. You know what that has meant to me.

Neil O'Connor

University of Dublin, Trinity College

July 2010

Abstract

Pervasive computing represents a vision of networked computers being distributed throughout our everyday environment in order to transparently provide services to people. The use of sensors enables the deployment of so-called *context-aware* pervasive computing applications that can perceive their operating context and alter their behaviour accordingly for the user.

Much work in the context-aware computing field focuses on simplifying application development by providing means for developers to define contexts of interest at design time. With these approaches correct application behaviour depends on the developer's knowledge. If the developer has an incorrect understanding of the deployment environment then the context definitions may be inaccurate. This is likely to be the case as applications are deployed in the real world, which is inherently unpredictable. Contexts are also typically defined relative to a fixed set of sensors or sensor types forseen by the developer. In the real world these sensors may not be available and may even be unsuitable for detecting the context in particular environments. An application cannot adapt context definitions that are fixed at design time to correct context-definition inaccuracy or sensor unavailability/unsuitability.

Some approaches in the context-aware computing field apply artificial intelligence (AI) techniques to reduce the dependency on the developer, by learning context definitions at run time. Existing approaches have the potential to adapt context definitions to the run-time environment, however they limit the application to a fixed set of sensors. They also depend on a discretisation layer that abstracts from raw sensor data to make the learning task feasible. This discretisation layer is fixed at design time and limits the resolution at which the application can learn its context definitions. It may therefore limit the accuracy of learned context definitions.

Our hypothesis is that context-aware applications in unpredictable environments are constrained by structures that are encoded at design time, and could more accurately infer their context if they self-configured their context definitions at run time. We characterise applications with this capability as *knowledge autonomous*. Knowledge-autonomous applications would extract meaning from sensor data and use this meaning to adapt their context definitions, thereby allowing them to use the most

suitable, available sensors.

We have identified two key requirements for knowledge autonomy in context-aware applications. Context definitions should be represented such that an application can independently interpret them, so they can be evaluated and corrected at run time. There should also be means to select the set of sensors used to distinguish contexts at run time, and evaluate sensor suitability relative to the application's goals.

To address these requirements we define the two-phase Knowledge Autonomy For Context-aware Applications (KAFCA) process, in which application-interpretable meaning for context is derived from application actions. Central to this process is the use of reinforcement learning to learn mappings from sensor data to actions at run time. As with other approaches that apply AI techniques this necessitates a discretisation layer. The first phase of the KAFCA process iteratively refines how data from individual sensors is discretised to improve the accuracy of the resulting context definitions. The second phase iteratively combines sensors based on their individual context definitions, and evaluates application performance while using different combinations. The most suitable set of sensors is identified from those available to a context-aware application.

We demonstrate that reinforcement learning is a suitable technique for learning the meaning of discretised sensor data. We show that the accuracy of context definitions can be significantly improved by refining the discretisation layer, subject to a trade off between accuracy and learning time. We also show that where user or environmental characteristics are sufficiently varied an application that selects its sensors at run time outperforms applications that use fixed sets of sensors.

Publications Related to this Ph.D.

- Neil O'Connor, Raymond Cunningham and Vinny Cahill *Self-adapting Context Definition.* In Self-Adaptive and Self-Organizing Systems (SASO), Boston, USA, July, 2007.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Context-aware applications deployed in a pervasive-computing environment are designed to identify their operating context from sensor data and adapt their behaviour appropriately. These applications may encounter unpredictable environmental and user characteristics when they are deployed, making it difficult to construct context definitions that are appropriate in all environments. In addition the set of sensors available to the application at run time may be unknown. Expected sensors may be unavailable, while suitable but unexpected sensors may be present. Sensor unpredictability increases the difficulty of defining application contexts.

This thesis investigates how context-aware applications attach meaning to sensor data so they can independently reason about their context definitions– be *knowledge autonomous*. The principal objective of this research is to define mechanisms to increase a context-aware application's accuracy at identifying its context thereby improving application performance. A secondary objective is to simplify the task of the developer when building this class of application by reducing the requirement for encoding expert knowledge.

This chapter introduces pervasive computing and its relationship with context-awareness, motivates the research, presents the contributions of the work and, finally, outlines a roadmap for the thesis.

1.1 Pervasive computing

Pervasive (or ubiquitous) computing is heralded as the next age of computing (Satyanarayanan, 2001; Weiser, 1991). Pervasive computing refers to the seamless integration of information technology (IT) in everyday life. The objective is to replace the current model of IT usage, where users actively

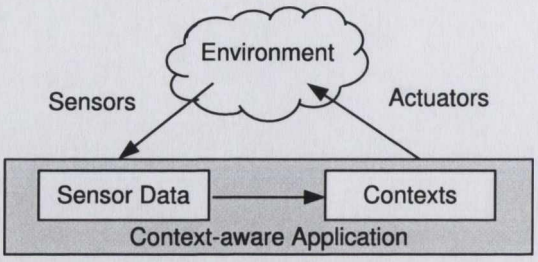

Figure 1.1: Structure of a context-aware application

perform tasks at dedicated computing devices, with a model where devices transparently assist users– providing information and services in an intelligent, unobtrusive manner. The user and their tasks should be the central focus rather than the technology.

While computing devices are not yet pervasive, we are progressing towards this vision. Personal digital assistants (PDAs), mobile phones, large-scale displays and digital cameras are commonly used devices that resemble those of Weiser's vision (Bell & Dourish, 2007). Mobile phone deployment, in particular, is widespread, and approaching saturation in places (Harris, 2007). These devices offer ever increasing processing power, storage, networking and functionality to replace that of the common desktop computer (Barton et al., 2006).

Although the hardware is becoming more widespread we are still some way from seamlessly providing useful services to the user. In Mark Weiser's own words– "applications are of course the whole point of pervasive computing" (Weiser, 1993). Context-aware computing addresses the challenge of building applications that provide these services.

1.2 Context-aware computing

Context-aware computing is a fundamental component in realising the vision of pervasive computing. Its objective is to increase application usability and effectiveness by taking environmental context into account (Baldauf et al., 2007). Gartner predict that "conditions will develop in 2009 through 2012 that will lead to mainstream adoption of context-aware computing" (Gartner, 2009).

Context describes the state of the environment in which an application operates. Dey's popular definition describes context as "any information that describes the relevant elements of a given situation" (Dey et al., 2001). Context-aware applications identify the context and adapt their behaviour accordingly without explicit user intervention (Schmidt et al., 1999).

Context data may come from a variety of sources including the physical environment, network statistics, device status and user profiles (Baldauf et al., 2007). We are particularly interested in

sensors that detect aspects of the physical environment, as this is the user interface for context-aware applications and key to transparent interaction. Figure 1.1 shows a context-aware application that gathers sensor data from the environment, infers its context, and alters how it affects the environment through actuators.

Obviously these applications depend on the availability of sensors in the environment. It is part of the pervasive-computing vision that sensors will be embedded in the environment to provide sensor data. There is already significant progress in this direction with laptop computers and mobile phones now featuring global positioning system (GPS) location sensors, accelerometers, cameras and microphones. In the future, sensors embedded in traditional systems may also be integrated into the pervasive-computing environment to provide sensor data to generic applications. Examples could include thermostats in home-heating systems, road-traffic cameras, and motion detectors on automatic doors. Increasing numbers and types of sensors may create new ways for applications to detect their context.

1.3 Motivation

Standard applications operate in a limited number of situations and a developer designs the application to operate appropriately in each situation (Kofod-Petersen, 2006). In contrast, context-aware applications operate in the real world, where each deployment environment is unique and the situations that arise are unpredictable. This unpredictability makes it more difficult to build applications that operate correctly.

Most research in context awareness focuses on technical and syntactic issues associated with storing, querying and interpreting context. These issues have been widely researched and surveyed (da Costa et al., 2008; Baldauf et al., 2007; Modahl et al., 2005). However the challenge of defining the "right" contexts for an application– contexts that cause appropriate adaptation at run time– has received little attention (Mikalsen & Kofod-petersen, 2005). In general this issue is ignored by making context definition a *knowledge-intensive* task. This places the responsibility for defining correct application contexts on the developer's expert knowledge of the application and environment. As Flanagan observes "an expert is needed to define contexts and user needs in those contexts" (Flanagan, 2005b). The accuracy of context definitions in knowledge-intensive approaches depends on the knowledge of the developer.

Unfortunately the developer cannot define how an application should react to every possible environmental situation since there are an unbounded number of situations in the real world (Banavar

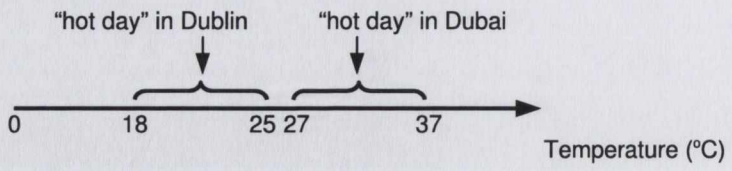

Figure 1.2: Context interpretation depends on the environment

& Bernstein, 2004). In order to build an application the developer must make assumptions about the run-time environment. Unpredictability in the real world means these assumptions may not be true, and we identify two problems that this causes for context definition.

1.3.1 Context misinterpretation

Correctly identifying the context depends on accurate interpretation of sensor data, however the correct interpretation may depend on the particular environment. Kofod-Petersen describes this challenge for context-aware applications: "The ability to know exactly what should be done, when, how and why are not necessarily well known when constructing the application" (Kofod-Petersen, 2006). For example an application designed to infer a "hot day" context in Dublin might interpret this from temperature readings in the range 18°C to 25°C, however the same temperature readings in Dubai would not indicate the "hot day" context (Fig. 1.2). Accurate interpretation of the context depends on the specific run-time environment in which the application is deployed.

Some researchers have applied learning techniques from artificial intelligence (AI) to address this challenge. The fields of AI and ubiquitous computing share an overlapping interest in interpreting and responding to the dynamics of the environment (Leahu et al., 2008). Context-definition approaches based on AI techniques learn about context in order to reduce their dependency on expert knowledge. Various techniques such as Bayesian networks, reinforcement learning and case-based reasoning have been applied and each has its own particular requirements and limitations when applied to learning context definitions. A common issue when using these techniques is that sensor data is too fine-grained to learn individual interpretations for all possible sensor values. As we discuss in Chapter 2 context-definition-learning approaches require a *discretisation* layer that maps from low-level sensor data to discrete states, to reduce the granularity at which sensor data is considered.

Context definitions are learned at the granularity offered by discrete states in the discretisation layer. Atkin states that "much of the art of problem solving (using AI techniques) depends on choosing the appropriate set of states" (Atkin & Cohen, 2000). If the discrete states do not capture important changes in the environment learned context definitions may be inaccurate and lead to incorrect appli-

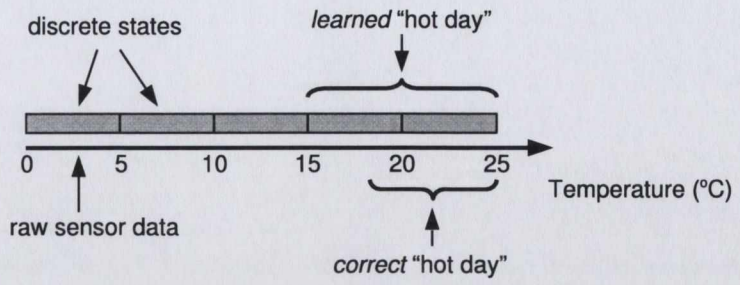

Figure 1.3: Poor abstractions lead to context misinterpretation

cation behaviour. For example consider the discrete states 0–5, 5–10, 10–15°C,... for a temperature sensor (Figure 1.3). In an environment where the context "hot day" is 18 to 25°C, the application (limited by the discretisation layer) would learn that 15 to 25°C is the "hot day" context. In this case temperatures between 15°C and 17°C are misinterpreted as "hot day" due to this context-definition inaccuracy.

The challenge of defining an appropriate set of discrete states is compounded in context-aware applications by unpredictability in the run-time environment. Discrete states that are suitable for learning in one run-time environment may not be suitable for learning in another. Existing context-definition-learning approaches define the discretisation layer using expert knowledge at design time and do not adjust it for the run-time environment (c.f. Chapter 2).

1.3.2 Sensor unavailability or unsuitability

Another potential issue caused by unpredictability in the run-time environment is sensor *unavailability*. Contexts are typically defined relative to a fixed set of sensors assumed to be available at run time, and in the absence of a required sensor the context cannot be inferred. A number of approaches (Dey et al., 2001; Bardram, 2005; Biegel & Cahill, 2004) treat the issue of sensor unavailability as a resource-discovery problem. In these approaches sensor types rather than specific sensor instances can be associated with an application at design time. Instances of these sensor types are discovered and connected to at run time. These approaches address syntactic issues of identifying and selecting sensor instances, however the issue of using unforeseen sensors is not addressed. Only sensors of exactly the expected types can be used. For example, a sensor that measures temperatures in °Fahrenheit or °Kelvin cannot be used to infer a "hot day" context by an application that only expects temperatures in °Celsius. The sensors produce equivalent information but only those of the expected type produce data that the application knows how to interpret. Existing approaches assume that at least one

instance of each expected sensor type will be available at run time and are unable to adapt if this is not the case (c.f. Chapter 2).

Environmental unpredictability may also affect sensor *suitability*. In many cases it is impossible for a developer to specify the particular combination of sensor data from which a context should be inferred (Banavar & Bernstein, 2004). A predefined set of sensors may not detect the relevant characteristics for inferring a context in a particular run-time environment. For example an application designed to infer a "meeting" context based on voice activity may not function correctly in a shared-office environment. In this environment simultaneous phone calls by multiple individuals could be misinterpreted as a "meeting". Existing approaches do not consider the suitability of sensors for inferring the context at run time (c.f. Chapter 2).

1.3.3 Hypothesis

These issues highlight the difficulty of defining contexts that are accurate in all run-time environments. Our hypothesis is that context-aware applications in unpredictable environments are constrained by elements of their context definitions that are fixed at design time, and could more accurately infer their context if they self configure their context definitions at run time. We characterise these applications as *knowledge autonomous*, as their context definitions do not contain knowledge from an expert. Knowledge-autonomous applications would independently attach meaning to sensor data, and this would facilitate context definitions that accurately interpret data from the most suitable sensors in the run-time environment.

1.4 Thesis objectives

This thesis addresses the challenge of defining the right contexts for context-aware applications. In Section 1.3 we identified ways in which an application's ability to identify its context could be affected by the unpredictability of its run-time environment. An application might misinterpret how sensor data indicates the context in different environments, or use sensors that are unsuitable for sensing the context. It might also be unable to identify the context if particular sensors become unavailable. We define two main objectives of this thesis to address these challenges.

Objective 1: Accurate run-time interpretation of context from sensor data

Applications should identify their context appropriately for the run-time environment in which they are deployed. To achieve this goal they should interpret sensor data at run time to identify contexts

correctly. The first objective of the thesis is to explore how existing approaches are limited in how they adapt application context definitions to the run-time environment, and how these limitations should be addressed to improve the accuracy of context definitions.

Objective 2: Suitable sensor selection from those available

The set of sensors that produce useful information for identifying contexts may depend on the characteristics of different run-time environments. The second objective of this thesis is to examine how existing approaches are limited in their ability to identify suitable sensors, and address these limitations to increase the autonomy of context-aware applications. This objective also addresses the issue of sensor unavailability, as applications would choose the most suitable sensors from those available.

1.5 Knowledge Autonomous Context-aware Applications

In this thesis we describe a process for Knowledge Autonomous Context-aware Applications called KAFCA. KAFCA addresses the reliance on expert knowledge that we identify as a limitation of context accuracy in existing approaches to context definition (Section. 1.3).

In a scenario where KAFCA is applied we envision a context-aware application deployed in an environment unforeseen by the developer. This environment may be very different from the envisioned run-time environment in terms of appropriate context definitions and available, suitable sensors. Initially the application learns how to infer its context from individual sensors. Contexts are characterised by the application adaptation they cause, so the application learns when it is appropriate to take particular actions. Contexts are defined on a discretisation layer to make learning feasible, and the discrete states in this layer are iteratively refined to increase the accuracy of dependent context definitions. Subsequent to learning context definitions for individual sensors the KAFCA process combines and evaluates different sets of available sensors, and the most suitable combination is identified. By adapting its context definitions at run time the application accurately identifies its operating context in its current environment.

Reinforcement learning (Sutton & Barto, 1998) is central to the KAFCA process as it is used to learn policies, which are mappings from sensor data to actions. A policy defines meaning for sensor data that the application can use to reason about its context definitions. KAFCA has two distinct phases and policies are used in both.

The first phase of the process learns accurate context definitions for individual sensors. Reinforcement learning necessitates a discretisation layer, however unlike other learning approaches to context

definition the discretisation process is not immutable. Instead it is refined at run time to improve the accuracy of context definitions. The refinement process is iterative. Reinforcement learning is used to learn policies, which in turn are used to identify *context-edge* locations– boundaries between different contexts. Discrete states near context edges are refined so that context edges can be identified with greater accuracy. This process is repeated to improve the accuracy of context definitions. This phase of KAFCA addresses the issue of context misinterpretation described in Section 1.3.1 by expressing contexts on discrete states that are adjusted for the run-time environment.

The second phase of KAFCA addresses the sensor unavailability and unsuitability issues described in Section 1.3.2. In general not all available sensors will be suitable for inferring the context so an informed search is used to identify potentially useful sensor combinations. Sensors are combined using the context definitions learned during the first phase of KAFCA. Application performance is measured while using different combinations and the results of the evaluation inform the search for other combinations. The most suitable sensors are iteratively identified and are selected as the run-time sensors for the application.

In the course of our evaluation we investigate the suitability of reinforcement learning as an approach for learning the meaning of discretised sensor data. Reinforcement learning requires that the application developer define a *reward model* that evaluates the effectiveness of application actions. There are two common reward-model types– long-term and immediate– and we investigate the implications of using each type when learning about sensor data.

We measure the effect of discrete-state refinement on context-definition accuracy, by recording their accuracy after each pass of the refinement process. We also record the learning overhead of each pass, and using this data we carry out a cost-benefit analysis of accuracy versus learning time.

Finally, we investigate the effect of sensor selection on application performance. The performance of an application that uses the KAFCA process to select sensors is compared to applications with different fixed sets of sensors. These applications are tested in environments where user and environmental characteristics vary, which creates the potential for sensor-suitability issues. Application performance is evaluated and compared across a number of application-specific metrics.

1.6 System model

KAFCA's scope includes context definition and sensor selection, but excludes other elements of context-awareness such as sensor discovery, description and communication, as well as actuator communication. These tasks are delegated to the MoCoA middleware Senart et al. (2006). MoCoA is a

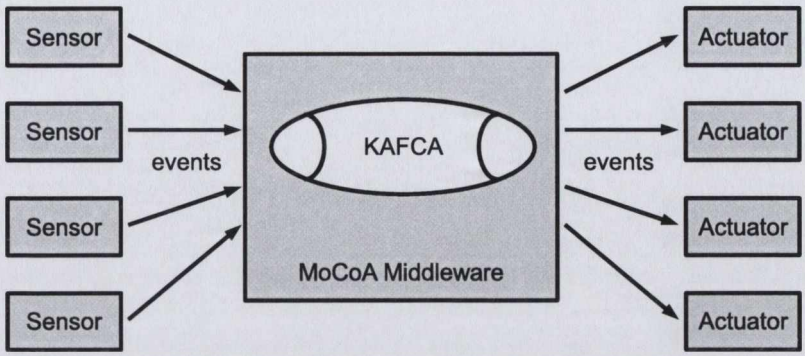

Figure 1.4: KAFCA within the MoCoA middleware

Value	Type	Description
Min	double	Minimum value that the sensor can produce (may be $-\infty$)
Max	double	Maximum value that the sensor can produce (may be ∞)
Precision	double	Maximum precision offered by the sensor
Initial-boundaries	[]double	Initial set of discrete-state boundaries

Table 1.1: Sensor meta data

middleware that provides a set of programming abstractions and services for building context-aware applications. In addition to the suitability of these abstractions for KAFCA this middleware was also chosen due to the author's prior experience with it. We discuss MoCoA in more detail in Section 4.3.

The overall system model is shown in Figure 1.4. MoCoA constructs applications using the sentient-object abstraction. Sentient objects are intelligent entities that extract and interpret context information from sensors to adapt their behaviour. The KAFCA process provides the logic within a sentient object by defining contexts and activating actuators in response to context changes. MoCoA presents sensors to the sentient object as software-event producers and actuators as software-event consumers. This abstracts away the complexity of interfacing with physical devices from KAFCA.

In addition to the data-delivery services offered by MoCoA the KAFCA process also requires that sensors be annotated with meta-data. These values are used to initialise discrete states for a sensor and control the degree of discrete-state refinement that occurs. Required meta-data values are listed in Table 1.1. The *Min* and *Max* values, along with the *Initial-boundaries* value are used to define the initial set of discrete states. The *Precision* value defines the limit to which discrete states can be refined. Various standards for sensor meta-data have been proposed. The most common standards are surveyed and compared in (Chen & Helal, 2008), with the most frequently used being SensorML (Open Geospatial Consortium, 2000) and IEEE 1451 (National Institute of Standards and Technology, 2005). All of these standards are suitable for representing this simple sensor meta-data.

1.7 Thesis assumptions

In order to scope the work a number of assumptions are made across the thesis. We summarise these assumptions in this section and they are discussed in more detail where necessary in the remainder of the thesis.

With respect to sensor data we assume that it is reliably delivered periodically to an application, and that sensor data from multiple sensors arrives, or at least can be reasoned about, in parallel. We also assume that sensors produce ordered sensor data, and that sensor data that cause similar the application to adapt its behaviour similarly are situated in close proximity to each other based on that ordering.

With regards to reinforcement learning we assume that the set of sensors that the reward model depends on are always available to an application, that the set of actions available to an application are known at design time, and that we can identify when sufficient learning has occurred based on the stability of learned policies. Although we expect unpredictable deployment environments we assume that the environment changes slowly enough to allow the application to learn stable policies using reinforcement learning.

1.8 Thesis contribution

To date a number of approaches to run-time context definition have been proposed. These approaches depend on underlying fixed structures that limit their ability to learn accurate context definitions and select suitable sensors. The context-definition learning approach described in this thesis contributes to the state of the art in context awareness by addressing the following issues:

- Existing definitions of context focus on describing context in terms of types of information: location, identity, objects and time. However these are human-interpretable structures that an application cannot process without interpretations defined by a developer. This thesis defines context in terms of application behaviour, which can be independently learned by the application at run time using AI techniques.

- Existing approaches to context definition rely to various degrees on structures that encode expert knowledge. These structures cannot be autonomously adjusted by the application at run time to correct context-definition inaccuracy. This thesis describes context definitions that are flexibly defined in terms of context edges, and a technique for refining definitions to improve their accuracy at run time.

- Existing approaches to context definition are restricted to sensor types that are defined at design time. However this limits their ability to adapt to sensor unavailability or unsuitability. This thesis describes an algorithm for selecting the most suitable application sensors from those available to improve application performance.

In summary, this thesis describes the development of a process for defining contexts that are accurate in the current run-time environment, and are inferred from the most suitable set of sensors available.

1.9 Thesis outline

The structure of the remainder of the thesis is as follows. Chapter 2 presents a survey of background material and related research in the field. It highlights the achievements and limitations of existing approaches. Chapter 3 introduces the main concepts of our approach to knowledge-autonomous, run-time context definition. Chapter 4 presents an implementation of the approach. Chapter 5 evaluates the effectiveness of the approach for a number of scenarios. Chapter 6 presents conclusions, open questions and opportunities for further research.

Chapter 2

State of the Art

Knowledge autonomy for context-aware applications is concerned with making context definitions independent of expert knowledge. This will allow applications adapt their context definitions to be accurate for the current run-time environment and infer the context from an unforeseen set of sensors, as discussed in Chapter 1. Context definitions define how context-aware applications reason about sensed context and their own behaviour. In the final step of this reasoning process the application makes a decision to adapt its behaviour, e.g., if $context_x$ do $action_y$. This step will almost certainly depend on other reasoning processes, e.g., to determine that the application is actually in $context_x$. We define a context definition as:

> All processes that alter or add interpretation to raw sensor data, between the point where a sensor generates data and the point where an application makes a decision to adapt its behaviour based on that data.

These holistic context definitions may encapsulate layers of processing and interpretation across multiple components of a context-aware application.

In our review of related work we compare four aspects of each approach, which reflect the objectives of the thesis outlined in Section 1.4. To evaluate the ability of approaches to accurately interpret context in different run time environments (Objective 1) we examine (1) the flexibility of their context definitions at run time and (2) the expert knowledge required to adjust those context definitions. The flexibility of definitions limits how they can be adjusted to correct inaccuracy at run time, and their dependency on expert knowledge limits how autonomously the adjustment can be made by the application. To evaluate the ability of approaches to select suitable sensors from those available (Objective 2) we examine whether they consider (3) the usefulness of sensors or sensor information

for identifying contexts at run time, and (4) how they address the issue of expected sensors being unavailable.

We distinguish between two general categories of context-definition approaches: *knowledge intensive* and *learned*. Most approaches to context awareness are in the knowledge-intensive category. Approaches in this category explicitly encode expert knowledge in context definitions dictating how an application identifies, interprets and adapts to context. In comparison a relatively smaller number of approaches are in the learned-context category. These approaches apply artificial intelligence (AI) techniques to learn context definitions. Learning approaches have the potential to learn beyond expert knowledge encoded in the application and adapt their context definitions independently.

The body of research in knowledge-intensive context-aware approaches is large. A number of best-practices have emerged and these have been integrated in frameworks, toolkits and middleware to aid application development. Due to the size and maturity of this body of research we limit our discussion to these generic approaches and how they approach context definition. The body of research in learned context is smaller and we structure our discussion around the underlying learning techniques. Each technique is illustrated with representative examples of approaches that apply it.

2.1 Knowledge-intensive context definition

Knowledge-intensive approaches to context definition are those that depend on expert knowledge to construct context definitions. Approaches in this category have progressed from initial standalone applications (surveyed in (Abowd et al., 1999; Pascoe, 1998)) to frameworks, middleware and toolkits that provide generic support for application development (surveyed in (Baldauf et al., 2007; Moore et al., 2007; Bolchini et al., 2007; Modahl et al., 2005; Strang & Linnhoff-Popien, 2004)). These generic approaches aim to simplify the process of application development by providing structures and services that are commonly needed in context-aware applications. Knowledge-intensive approaches can be further categorised based on whether they support changes to context definitions at run time or only at design time. This has obvious implications for the adaptability of their context definitions to the run-time environment.

2.1.1 Design-time approaches

D*esign-time* approaches are a subset of knowledge-intensive approaches to context awareness (da Costa et al., 2008; Banavar & Bernstein, 2004). These approaches encode context definitions in the application at the design stage of its life cycle and the definitions are immutable at run time. A large number

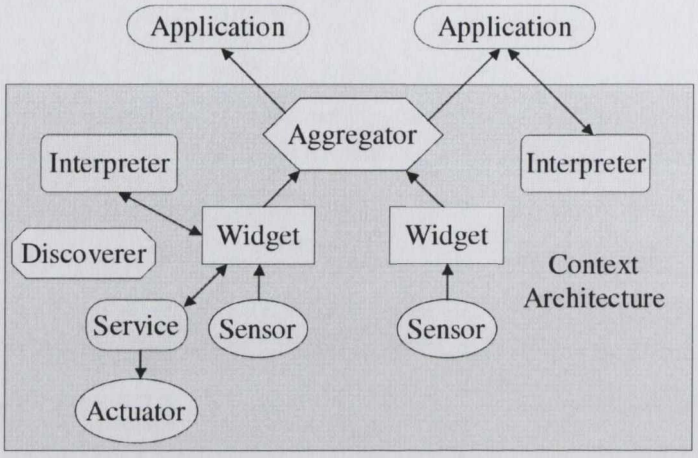

Figure 2.1: Context Toolkit components and relationships from (Dey et al., 2001)

of approaches belong to this category.

2.1.1.1 Context Toolkit

The Context Toolkit (Dey et al., 2001) is a conceptual framework and toolkit developed at the Georgia Institute of Technology. Its main aim is to free developers from low-level issues associated with acquiring sensor data so they can focus on high-level application behaviour. The approach is inspired by toolkits for graphical user interface (GUI) development that insulate the developer from presentation issues using widgets. The Context Toolkit encapsulates context-awareness concerns using five components. *Context widget* components encapsulate the complexity of interfacing with individual sensors and provide application access to sensor data. *Interpreter* components process and transform data from a widget or widgets into higher-level context knowledge. *Aggregator* components gather logically-related knowledge from multiple context widgets for easy access by applications. *Service* components encapsulate actuators that the application can activate to affect the environment. *Discoverer* components support reuse of components by maintaining a registry of available components. All components execute independently of each other and may be reused by multiple applications. Fig. 2.1 shows an example configuration where two applications connect to multiple components.

At design time the developer specifies which components are required for a particular application. Where necessary the developer creates components that are not already available. At run time the application uses a discoverer component to find required widget, interpreter, aggregator and service components. Component lookups are performed at discoverers using either a specific name (white pages lookup) or service description (yellow pages lookup).

Analysis Context definitions in the Context Toolkit are spread across components used by an application as well as the application itself. Context widgets can process sensor data to identify low-level contexts, e.g., determine the activity in which a user is engaged. Interpreter widgets process data from context widgets to determine high-level contexts, e.g., identify that a meeting is occurring based on the number of users and sound level in a room. The application itself defines how service components should be invoked in reaction to particular contexts. These processes combine to form the context definitions of the application. In the Context Toolkit components and applications are defined at design time and there is no facility for adjusting their encapsulated context-definition logic at run time. Therefore the issue of context misinterpretation and the associated inaccuracy in context definitions cannot be addressed by this approach at run time.

The Context Toolkit encapsulates individual sensors within context widgets, and applications are configured to use a particular set of widgets to determine their context. The yellow-pages lookup service offered by discoverer components allows applications to adapt to the unavailability of particular instances of their required sensors. This partially addresses the issue of sensor unavailability however it relies on at least one instance of each required sensor being available. The issue of selecting sensors that are suitable for detecting the context in the current run-time environment is not considered in this approach.

This approach depends entirely on the developer's ability to encode context definitions, therefore applications based on this approach have no knowledge autonomy.

2.1.1.2 Java Context-Awareness Framework (JCAF)

The Java Context-Awareness Framework (JCAF) (Bardram, 2005) is a Java-based infrastructure and application programming interface (API) for building context-aware applications. Its goal is to provide a simple framework that developers can extend to support particular applications. There are two core parts to the framework: a programming model and a run-time architecture. The JCAF programming model enables the developer to build applications for deployment in the JCAF infrastructure. It provides a minimal set of Java interfaces and classes for generic support of context modeling. A context model defines the real-world entities that are of interest to an application. Context modeling in JCAF is carried out by building object-oriented models in Java. The core modeling interfaces are *Entity*, *Context*, *Relation* and *ContextItem* (Fig. 2.2). These interfaces are implemented and extended by the developer to define a model of context for an application. Each context model is managed by a *context service* in the run-time architecture.

The run-time architecture of the JCAF is three-tiered. The first tier contains applications, the

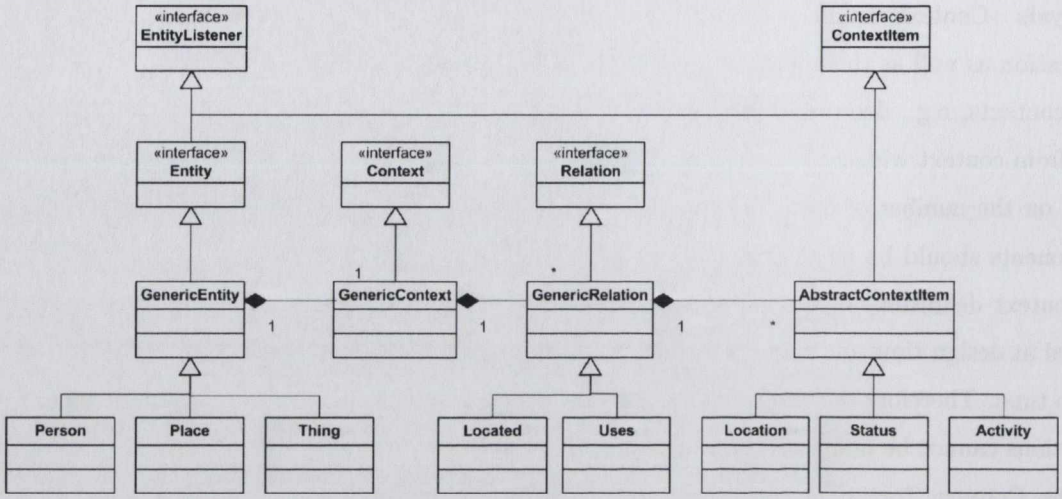

Figure 2.2: JCAF context model from (Bardram, 2005)

second contains context services, and the third contains sensors and actuators. Context services gather sensor data from sensors, relate it to entities in their managed context model and inform applications of resulting context changes. The JCAF uses an event-based publish/subscribe approach to communicate between applications and context services. Type-based subscription is supported so applications can subscribe to context changes of a particular type from any context service. Applications encode knowledge of how to respond to contexts, and their reactions are propagated by context services to actuators. The run-time architecture is designed to be modifiable and extensible at run time by supporting addition, deletion and modification of context services, sensors, actuators and applications.

Analysis Context definitions in the JCAF span context services, context models and applications. Context services interpret sensor data to identify its effect on context models and propagate context changes to applications. Applications interpret and react to these context changes. The JCAF run-time architecture is modifiable at run time however changes can only be made at the granularity of components. There is no run-time mechanism for changing the internal structure of context services, context models or applications so the issue of context misinterpretation cannot be addressed by this approach at run time.

The JCAF run-time architecture uses the publish/subscribe paradigm to communicate between applications and context services. It supports type-based subscription so applications can subscribe to all context information of a particular type. This partially addresses the issue of sensor unavailability as the application does not depend on a particular context-service (and hence sensor) instance, however

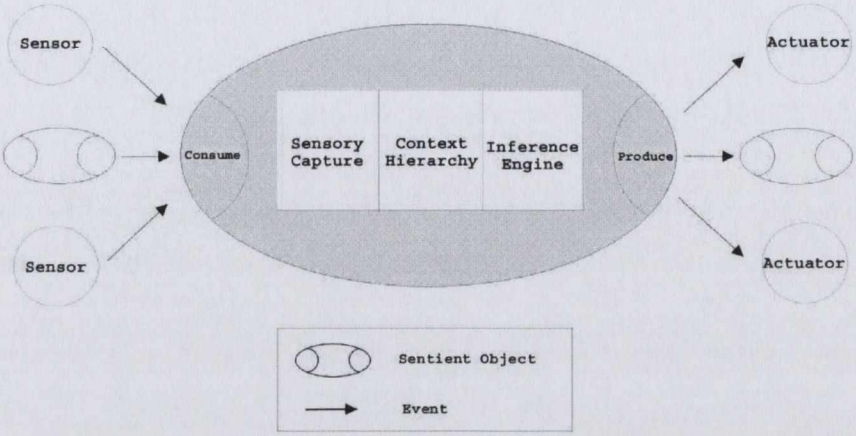

Figure 2.3: Sentient Object Model from (Biegel & Cahill, 2004)

it relies on at least publisher for each type of context information being available. The issue of sensor unsuitability is not considered in the JCAF.

As in the Context Toolkit this approach relies entirely on the developer's ability to encode context definitions, therefore the resulting applications have no knowledge autonomy.

2.1.1.3 Sentient Object Model

The Sentient Object Model (SOM) (Biegel & Cahill, 2004) is a framework that supports the development of mobile, context-aware applications. Its goal is to make this class of application easier to design, prototype and test so that developers and end users are empowered to build their own applications. To achieve this it provides a visual programming tool for *sentient objects*, which reduces the need to write complex code. Sentient objects are entities with interfaces to sensors, actuators and other sentient objects. The SOM applies an event-based communication approach based on the publish/subscribe paradigm to connect sentient objects to sensors and actuators at run time. A sentient object consumes software events from sensors and other sentient objects, performs some internal processing of these inputs, and produces software events that influence actuators and other sentient objects (Fig. 2.3).

A sentient object has three internal reasoning components. The *sensory-capture* component interprets low-level sensor data to derive higher-level context data. Bayesian networks may be used to fuse sensor data from multiple sensors. The *context-hierarchy* component manages a hierarchical set of contexts in which the application may exist. The hierarchy is based on the Context-Based Reasoning (CxBR) paradigm (Gonzalez & Ahlers, 1998), and encapsulates knowledge about possible actions in

contexts and consequences of those actions. The *inference-engine* component applies conditional rules to reason about the context based on context data derived in the sensory capture component. It is also responsible for reasoning about adapting application behaviour.

Analysis Context definitions in the SOM span all three internal components of a sentient object, and may span other sentient objects that provide input. The sensory-capture component derives context data from sensor data. The context-hierarchy encapsulates knowledge of potential contexts and actions. The inference-engine encapsulates knowledge in rules to reason about contexts and actions. There is no support for adjusting context definitions at run time to address the issue of context-misinterpretation.

Sentient objects and sensors in the SOM are loosely coupled due to the underlying event-based, publish/subscribe communication model. Similar to the JCAF this creates the potential for sentient objects to use any sensors that publish sensor data of the expected type, and partially addresses the sensor-unavailability issue. The issue of sensor unsuitability is not addressed.

This approach relies entirely on the developer to define context definitions within sentient objects at design time, therefore its applications have no knowledge autonomy.

2.1.1.4 Summary

Knowledge-intensive approaches that do not facilitate adjustments to context definitions at run time obviously cannot adapt their context definitions to the run-time environment. Any changes to the structures underlying their context definitions require a return to the design phase. These approaches can only address the context misinterpretation issue described in Section 1.3.1 by "undeploying" the application, editing its context definitions, and redeploying. This process may have to be repeated many times if multiple instances of an application in different environments require unique context definitions. The process of defining accurate context definitions may also be iterative if misinterpretations of the context are not completely understood prior to redesign.

Tsang argues that design-time approaches can facilitate run-time changes to context definitions through user preferences (Tsang, 2009). Knowledge-intensive approaches can treat user preferences as a sensor input. For example, in the Context Toolkit a context widget could represent the user's food preference e.g. Italian, Mexican or Chinese. The preference value can act like a switch on how the application reasons about context. It effectively enables and disables different sets of context definitions at run time. For example in Fig. 2.4 the food-preference value changes how the application acts in $context_A$ from $action_X$ to $action_Y$. The preference is a part of the context definition as it

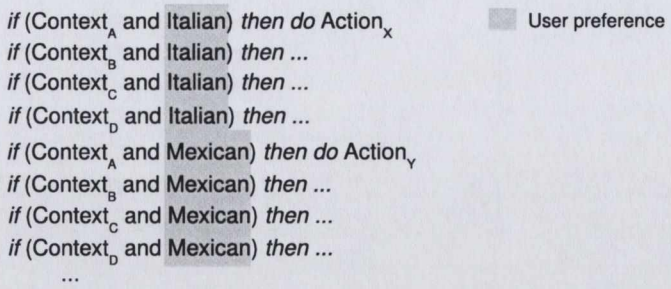

Figure 2.4: User preferences in context definitions

influences how the application reasons about context. However each preference switch at run time must be foreseen and encoded at design time and there is no facility for adjusting how preferences influence decisions at run time. Therefore context definitions that include user preferences do not address the issue of context misinterpretation at run time.

In these approaches interpretations of sensor data are encoded in context definitions and applications are therefore limited to sensors for which they have interpretations. Context definitions based on other sensors must be encoded using expert knowledge and require a return to the design phase.

These approaches define loosely-coupled relationships between applications and sensors, and lookup services are used to locate sensors at run time. Lookups based on sensor type rather than specific sensor instances can handle the failure of individual sensors. However in these approaches the interpretations of sensor data are encoded in their context definitions and applications are limited to the types of sensors that they can interpret. Therefore these approaches provide limited support for the issue of sensor unavailability as they require that at least one instance of each required sensor is available. The issue of sensor unsuitability for detecting the context in particular environments is not addressed.

These knowledge-intensive approaches are completely dependent on the expert knowledge of the developer when building applications therefore they exhibit no knowledge autonomy.

Other examples of knowledge-intensive approaches that are similarly limited to design-time context definition include SOCAM (Gu et al., 2004), Context Fabric (Hong & Landay, 2004), Context Shadow (Jonsson et al., 2003) and Hydrogen (Hofer et al., 2003).

2.1.2 Run-time approaches

Knowledge-intensive *run-time* approaches facilitate changes to context definitions at run time. These approaches are motivated by the need for personalisation of context-aware applications, as users may have diverse requirements (Henricksen et al., 2006) and these requirements can only be estimated at

run time (Korpipaa et al., 2005).

2.1.2.1 PACE

A framework for context-aware computing was developed as part of the Pervasive Autonomic Context-aware Environments (PACE) project (Henricksen et al., 2006). The goal of the framework is to integrate user preferences with context as a basis for flexible adaptation decisions at run time. The framework is divided into layers to support context awareness and user preferences (Fig. 2.5). The *context-gathering* layer uses event-based communication to gather data from sensors. This data may be processed into higher-level context data using interpreters and aggregators. The *context-reception* layer translates inputs from the context-gathering layer into a fact-based representation for the layer above. The *context-management* layer maintains a set of context models that are defined in a predicate logic. These models are populated with facts from the context-reception layer. The *query* layer provides an interface for applications and the preference-management layer to query context models. The *preference-management* layer stores repositories of preferences and evaluates preferences on behalf of the programming toolkit. Finally the *programming toolkit* lies between applications and the preference-management layer. It provides methods for *preference-based branching*, which is a process of evaluating different contexts based on the user's preferences.

Users express their preferences by assigning a score to behaviour choices for a particular context. Preferences are expressed as context-score pairs, where a score is a numeric value in the range 0 to 1 (larger scores imply increased preference for an option) or a special character ($\overline{\wedge}$– option must be selected, $\natural$– option must be vetoed, $\perp$– indifference to option). Fig. 2.6 shows some example preferences for a context-aware communication tool (Henricksen et al., 2006). The preference name (*p1–p4*) is shown on the left and the context and scoring expressions for each preference are preceded by the keywords *when* and *rate* respectively. For example *p2* states that *when* a synchronous channel is requested *and* the user is occupied *and* the purpose is not urgent then the request is forbidden (*rate* $\natural$).

This preference format is not directly exposed to the user. Instead the authors suggest that applications include an interface for customising application-specific preferences. The programming toolkit compares the scores assigned to different behaviour choices in a particular context and selects the highest scoring behaviour to carry out.

Analysis The PACE framework provides a model for customising context-related preferences at run time. Context definitions within the framework span the context-gathering, context-reception,

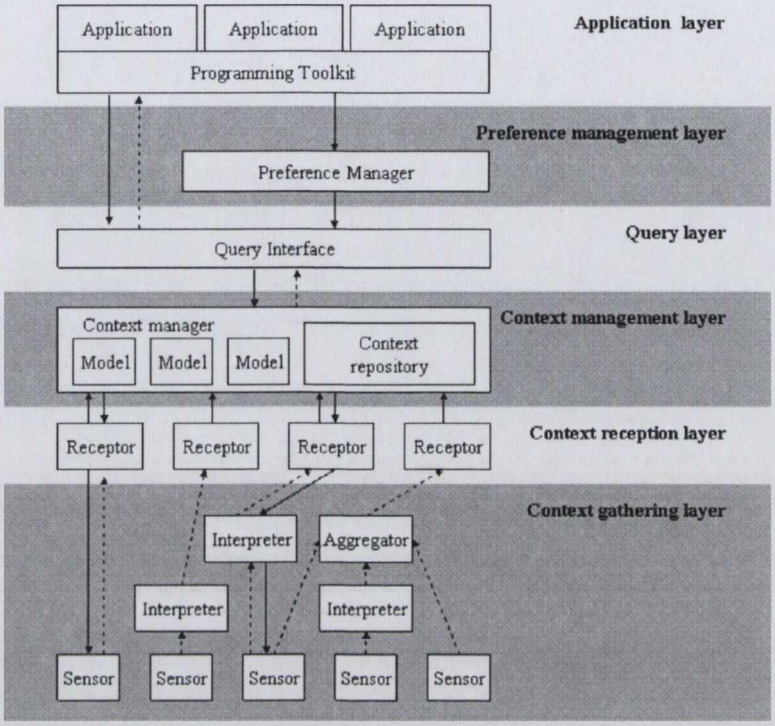

Figure 2.5: The PACE framework from (Henricksen et al., 2006)

$$
\begin{aligned}
\texttt{p1} &= \texttt{when } \mathit{SynchronousMode}(\mathit{channel}) \wedge \neg \mathit{CanUseChannel}(\mathit{callee}, \mathit{channel}) \\
&\quad\ \texttt{rate } \natural \\
\texttt{p2} &= \texttt{when } \mathit{SynchronousMode}(\mathit{channel}) \wedge \mathit{Occupied}(\mathit{callee}) \wedge \neg \mathit{Urgent}(\mathit{priority}) \\
&\quad\ \texttt{rate } \natural \\
\texttt{p3} &= \texttt{when } \mathit{Urgent}(\mathit{priority}) \wedge \mathit{SynchronousMode}(\mathit{channel}) \\
&\quad\ \texttt{rate } 1 \\
\texttt{p4} &= \texttt{when } \mathit{Urgent}(\mathit{priority}) \wedge \neg \mathit{SynchronousMode}(\mathit{channel}) \\
&\quad\ \texttt{rate } 0.5
\end{aligned}
$$

Figure 2.6: PACE preferences for a context-aware communication tool from (Henricksen et al., 2006)

context-management layers and preference-manager layers, as each of these layers carries out some interpretation of sensor data or affects reasoning about context in some way. The framework empowers the user to adjust the user-preference element of context definitions at run time by assigning scores to preferences. The other elements are fixed in the application at design time and may introduce inaccuracy into the context definitions that cannot be corrected at run time. As a result the PACE framework has limited ability to address the issue of context misinterpretation at run time.

The PACE framework loosely couples sensors and applications as it uses an event-based communication model. This creates the potential for applications to use any sensors that publish events of

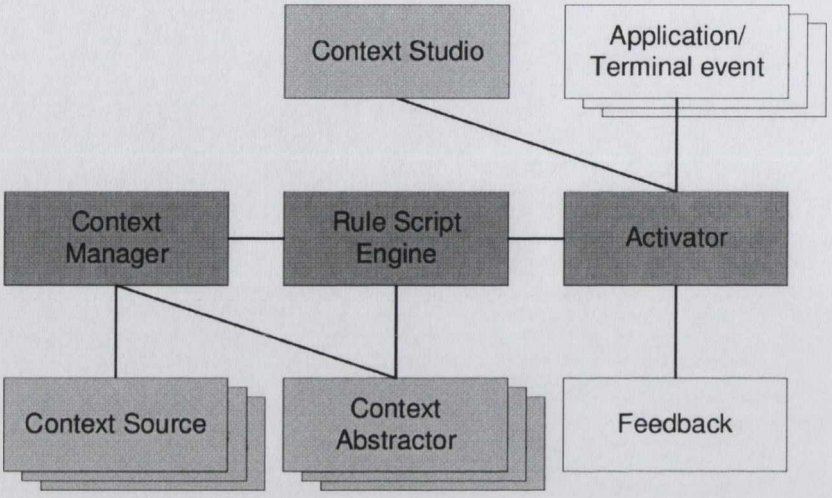

Figure 2.7: Context Studio from (Korpipaa et al., 2005)

the required type. The sensor unavailability issue is partially addressed as the application needs at least one sensor of a required type to be available. The issue of sensor unsuitability is not addressed.

This approach depends entirely on expert knowledge from the developer and user therefore the applications have no knowledge autonomy.

2.1.2.2 Context Studio

The Context Studio (Korpipaa et al., 2005) is a tool for customising context-aware applications. Its goal is to empower end users to create their own context-aware solutions using a GUI. Instead of defining context definitions at design time the Context Studio provides contexts and actions to the user so they can define context definitions at run time. The set of available contexts and actions is defined in an ontology at design time, and the GUI enables the user to bind contexts to application actions using *condition-action* rules at run time. Rule conditions can combine multiple pieces of context information using logical operators such as and, or, not .

A context framework handles the background monitoring of contexts and rules (Fig. 2.7). *Context-source* components wrap context-information providers such as sensors. *Context-abstractor* components perform operations on sensor data to infer higher-level context information. Sensor data is transformed by context source and abstractor components to a set of contexts defined by an ontology. The *context-manager* component receives and stores all context data and provides a uniform interface for accessing it. The *rule-script-engine* component monitors the contexts in user-defined rule conditions and activates application actions through the *activator* component. The *feedback* component is

an actuator that informs the user of application actions using vibration or sound.

Analysis The Context Studio provides end users with an interface for defining their own context definitions. Context definitions in the Context Studio span context sources and context abstractors, the context ontology, and user rules. Only the user rules are adjustable at run time through the Context Studio. Therefore it has limited ability to address the issue of context misinterpretation at run time.

The application ontology limits the context information, and hence the set of sensors, available for defining rules. There is no support for adjusting the ontology or the underlying set of context sources and abstractors at run time. This means the Context Studio cannot adapt to sensor unavailability. The issue of sensor unsuitability is not addressed either. This approach has the potential to be extended such that the user can integrate new sources of sensor data and context at run time. This would enable the approach to address both these issues.

Again, this approach is entirely dependent on the expert knowledge of the developer and user therefore it has no knowledge autonomy.

2.1.2.3 Summary

Knowledge-intensive approaches that adjust context definitions at run time are better able to address context-misinterpretation issues than purely design-time approaches, as the user can correct inaccuracies as they arise. However the described approaches facilitate limited changes to context definitions. Context definitions still depend on elements that are statically defined at design time and may introduce inaccuracy that cannot be corrected at run time.

As with knowledge-intensive design-time approaches these approaches may define loosely-coupled relationships between applications and sensors. This allows the application adapt to the unavailability of particular sensor instances as long as at least one sensor of the required type is available. This partially addresses the sensor unavailability issue. The issue of sensor unsuitability is not addressed. As noted above there is potential to extend knowledge-intensive approaches such as these to enable the user to integrate now sensors at run time. This extension would address both of these issues.

Such approaches depend on the developer and user to explicitly define and adjust context definitions using expert knowledge therefore they exhibit no knowledge autonomy.

2.2 Learned context definition

This section describes approaches to context-definition that apply learning techniques. These are approaches that gather knowledge beyond that explicitly encoded in context definitions by developers at design time or users at run time. Learning techniques have been applied to context-awareness for almost as long as context-awareness has been studied (Clarkson & Pentland, 1998) however context-learning approaches are not as widespread as knowledge-intensive approaches.

Leahu et al. summarise the limitations of knowledge-intensive approaches and argue the case for learning in context awareness, drawing parallels between knowledge-intensive approaches and classical AI approaches (Leahu et al., 2008). They state that the typical approach in classical AI was to identify real-world entities, represent them using symbolic representations and use rules to reason about the symbols. However AI researchers discovered that there are "serious technical limitations to the goal of building complex, reliable, and dynamically relevant world models". Knowledge-intensive approaches to context definition are also based on world models and will encounter the same limitations (Leahu et al., 2008). In a similar argument Bell and Dourish state that most pervasive-computing research is "oriented around a conception of the world as orderly and homogenous" but that "the world in which ubicomp systems are currently deployed is messy and heterogenous and is likely to stay that way" (Bell & Dourish, 2007).

Learned context-definition approaches cite two main motivations: *personalisation of user context definitions* and *context-definition accuracy at run time.* Flanagan argues that useful context-awareness requires user personalisation, but that users are restricted by the complexity of manually personalising context definitions at run time (Flanagan, 2005a). Context-definition learning addresses this concern by removing the requirement for manual personalisation. Kofod-Petersen states that predicting accurate context definitions for the run-time environment is not always possible at design-time (Kofod-Petersen, 2006). Context-learning approaches address this concern by learning accurate context definitions based on the run-time environment. The context-definition accuracy argument is a more general case of the personalisation argument, as personalised context definitions address accuracy for individual users whereas context-accuracy in general may apply to applications with one, or many, or even no users.

In order to compare learned context-definition approaches we first describe different learning types: supervised, unsupervised and reinforcement learning. We also describe a common discretisation architecture that is shared by context-learning approaches and explain its motivation. Finally we discuss different learning techniques that have been applied to learning context definitions: Bayesian networks, artificial neural networks, reinforcement learning, case-based reasoning and data mining.

2.2.1 Supervised, unsupervised and reinforcement learning

Learning techniques can be characterised by *how* they learn. Russell and Norvig distinguish three types of learning: supervised, unsupervised and reinforcement learning (Russell & Norvig, 2003). *Supervised-learning* techniques learn a general function for mapping inputs to outputs from training data. Training data consists of example inputs and the corresponding outputs provided by external supervisors. *Unsupervised-learning* techniques learn how input data is organised based on similarities between input data, and without any requirement for training data. *Reinforcement-learning* techniques learn the usefulness of actions using feedback from the environment. They use trial-and-error to identify the best action to take in different situations.

Analysis Supervised-learning techniques require training data, which creates a dependency on expert knowledge. Training examples may be difficult to obtain and the accuracy of the general function depends on the accuracy of the training examples (Veeramachaneni et al., 2005). Supervised-learning techniques may also encounter an *incompleteness* problem where training data does not adequately represent general data (Tapia et al., 2006). Another issue with supervised-learning approaches is that they can only generalise about data for which they have training examples. In the case of learning context definitions this means that only sensors for which there is labeled training data can be used. This obviously limits their ability to select suitable sensors at run time.

Unsupervised-learning techniques learn organisational structures within input data however the relevance of those structures to the application is unknown. A purely unsupervised approach cannot learn appropriate application behaviour because it has no information about what constitutes a correct action (Russell & Norvig, 2003). In the case of learning context definitions this might mean that a set of application contexts can be identified, but how they affect application behaviour is unknown.

Reinforcement-learning techniques use feedback from the environment to inform them of the usefulness of their actions. Expert knowledge is required to encode interpretations for feedback and define what *good* actions are. These techniques also use a trial-and-error approach to learning that may be unsuitable for some applications, e.g., safety-critical systems.

2.2.2 Common discretisation architecture

As discussed in Chapter 1 context data may come from a variety of sources (Baldauf et al., 2007), but we are particularly interested in sensors that sense aspects of the physical environment. Sensors in this class may be capable of producing thousands of unique sensor values depending on their precision. This presents a challenge to learning techniques as the complexity of the problem scales linearly with

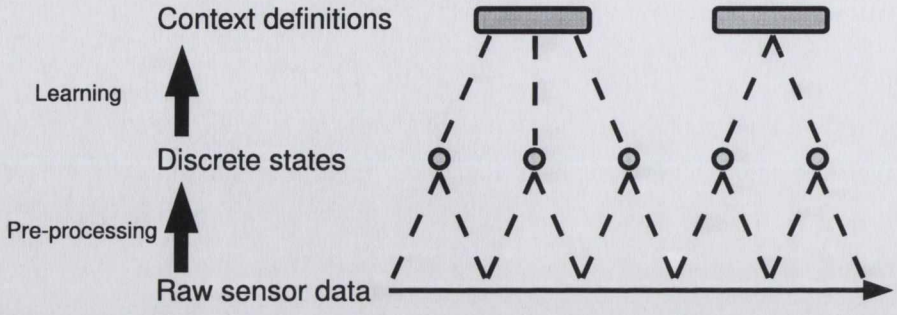

Figure 2.8: Common discretisation architecture

the number of states they consider (Sutton & Barto, 1998). Laerhoven and Cakmakci observe that if raw sensor values are used as inputs for learning techniques the performance of the application would suffer (Van Laerhoven & Cakmakci, 2000).

As a solution for learning about context definitions it is considered reasonable to "fix a set of basic features that are extracted from the observed data" (Himberg et al., 2003). Fig. 2.8 demonstrates how a preprocessing step translates multiple raw sensor-data values to a layer of representative discrete states, so that the number of inputs to the learning technique is manageable. Context definitions are learned on top of this layer. This architecture is common to many context-definition learning approaches and is variously referred to as discretisation (Brdiczka et al., 2006), quantization (Battestini & Flanagan, 2005), feature extraction (Ma et al., 2003) and cue extraction (Van Laerhoven & Cakmakci, 2000) in the literature.

Analysis Discretisation of sensor data is necessary in order to facilitate learning techniques, however it may affect the accuracy of learned context definitions. Discretisation causes information loss relative to the original source, which may result in important changes in the environment going unnoticed (Park et al., 2006; Albinali et al., 2007). Context definitions are learned on top of the discretisation layer and can only capture environmental changes that are captured by the discrete states. When important changes are not captured during discretisation the result is context-definition inaccuracy and therefore context misinterpretation (Section 1.3.1). The challenge for context-definition-learning approaches is to define appropriate discrete states for sensor data such that information lost during discretisation is not important to the application.

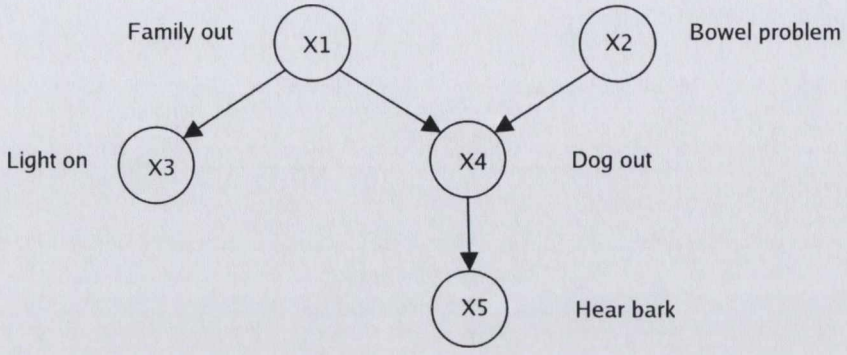

Figure 2.9: An example Bayesian network from (Charniak, 1991)

2.2.3 Bayesian networks

This section describes approaches that apply Bayesian networks to learning context definitions. A Bayesian network is a probabilistic model for reasoning. It is a compact representation of a joint probability distribution over a set of variables (Callan, 2003), i.e., the probabilities that two or more events occur at the same time. In a Bayesian network each node represents a variable. Each variable has a set of possible discrete states and a *conditional probability table* with associated probabilities of each state occurring. Links between nodes in the network encode conditional dependencies between variables, so the network and tables provide a decomposed representation of the joint probability distribution for the variables. In practical terms, given evidence of the state of some variables in the network a Bayesian network reasons about the likely state of other variables.

A classic example of a Bayesian network is illustrated in Fig. 2.9 (Charniak, 1991). The network defines the causal relationships between a family being out (*X1*), whether the family dog has a bowel problem (*X2*), whether a light in the home is on (*X3*). The links between nodes represent interdependencies. By observing the values of the leaf nodes (light on, hear bark) probabilities can be calculated for the other nodes in the network.

The network structure and the conditional-probability tables for nodes can be either hand-defined using expert knowledge or learned using supervised learning (Tapia et al., 2006). *Parameter learning* is used to learn the joint probability of states, i.e., the probability of a state occurring at one node given the states of other nodes. *Structural learning* is used to learn the links between nodes in the network, i.e., their causal dependencies.

A simple Bayesian network called a *naïve Bayes classifier* is often used in context-learning approaches. This network assumes that the leaf variables of the network are independent of each other.

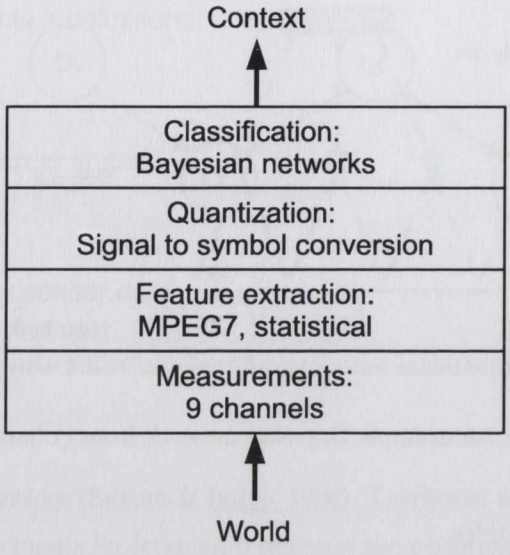

Figure 2.10: Korpipaa et al. context-representation layers from (Korpipaa et al., 2003)

Each variable independently contributes to the probability of higher-level variables (contexts) and less training data is required to learn these probabilities. *Hidden Markov models* are another commonly-used technique that represent temporal probabilities using sequences of Bayesian networks (Russell & Norvig, 2003). They are often used for temporal pattern recognition, e.g., speech.

2.2.3.1 A naïve Bayes classifier approach

Korpipaa et al. apply a naïve Bayes classifier to classify the contexts of a mobile-device user (Korpipaa et al., 2003). Their approach combines sensor data from accelerometers, a microphone, a thermometer, a light sensor, a humidity sensor and a skin-conductivity sensor. Sensor data is processed through a series of context-representation layers before being classified in contexts by the naïve Bayes classifier (Fig. 2.10). The *measurements* layer gathers sensor data from nine application sensors. The *feature-extraction* layer applies the MPEG-7 standard to extract 47 discrete sound features, and also performs statistical processing on data from other sensors to extract discrete features such as walking, running, indoor lighting, and sunlight. The *quantization* layer maps extracted features to representative symbols for processing by the naïve Bayes classifier. The *classification* layer identifies active contexts from a pre-defined set of thirteen possible contexts (inside, outside, speech, car, elevator etc.) using a naïve Bayes classifier.

In this approach the network structure of the naïve Bayes classifier is pre-defined so no structural learning is required. The network probabilities are learned using supervised parameter learning.

Training data is gathered by placing the device and sensors in known contexts and periodically recording the set of symbols from the quantization layer. Each periodic set of recorded symbols is manually labeled by the developer with the correct context. The network probabilities are learned by finding the maximum a posteriori (MAP) estimate for the training data. The posterior probability is the probability that is computed after evidence has been observed (Callan, 2003). Learning is carried out before the application is deployed so the network probabilities are static at run time.

Analysis This approach learns the probabilities of different contexts given evidence from sensors using supervised learning. Learning occurs at design time therefore the accuracy of learned context definitions depends on training data being representative of the run-time environment. Context definitions are not adjusted at run time.

Sensor data is processed at the feature-extraction and quantization layers before learning occurs in the classification layer. Learned context definitions span all of these processes but only the network probabilities are learned. The other processes are fixed and may introduce inaccuracy in context definitions that the learning process cannot address. The set of sensors in the measurements layer is fixed by the developer at design time and there is no facility for adapting to unavailability of sensor instances. The issue of sensor unsuitability is not addressed either.

This approach depends on expert knowledge to define the feature extraction and quantization layers, and also to correctly label the training data with contexts. However it removes the requirement for the developer to explicitly define the entire context definition therefore it exhibits some knowledge autonomy.

2.2.3.2 A hidden Markov model approach

Smith et al. describe a framework for identifying user context based on environmental noise (Smith et al., 2006; Ma et al., 2003). Rather than focus on speech as evidence of context the approach interprets complex sounds from a variety of sources such as air conditioning, car motors and keyboard clicks. They apply a hidden Markov model to process sequences of sound data and identify high-level contexts. Sensor data passes through two phases before being processed by the hidden Markov model. In the *preprocessing* phase audio data is segmented in clips, and in the *feature-extraction* phase features of each clip are extracted into mel-frequency cepstral coefficients (MFCCs). An MFCC represents the power spectrum of a particular sound. The feature-extraction phase identifies any of thirty-nine potential MFCCs in an audio clip, and these become the inputs to the hidden Markov model.

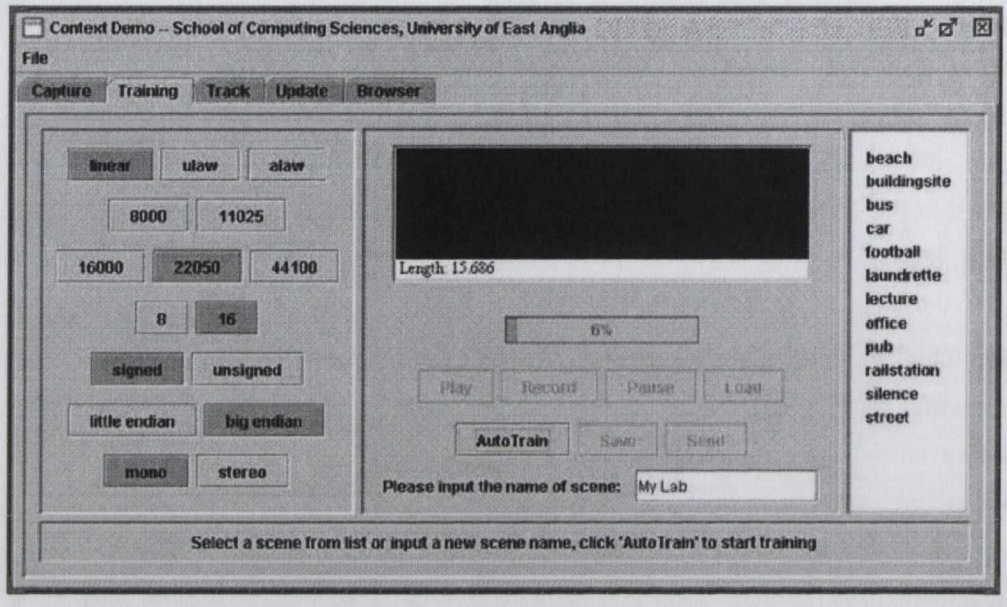

Figure 2.11: Smith et al. runtime training-data interface from (Smith et al., 2006)

Of particular interest in this framework is their approach to run-time learning. Initial training data for supervised learning is gathered prior to deployment and used to learn default network probabilities. The authors recognise the need to adapt the application to its run-time environment so they facilitate gathering new training data at run time. The user is responsible for labeling this data with the context in which it is gathered. The initial training data and run-time training data are combined and new network probabilities are learned. Gradually the initial training data is replaced by run-time data so that network probabilities reflect the run-time environment. Fig. 2.11 shows the user interface for recording training data at run time. The user specifies a name for the new context or selects the name of an existing context that needs retraining. An audio sample is recorded, processed, and new network probabilities for the specified context are learned.

Analysis This approach applies supervised learning and a hidden Markov model to learn context definitions. Unlike other Bayesian network approaches to context definition it recognises that the run-time environment may be unpredictable and facilitates gathering training data at run time. By gathering and labeling data at run time it addresses the issue that supervised learning is only as accurate as the training data. Other approaches artificially create a context in which to gather and label sensor data, which may not match the real world. Here the user labels the training data with the context that they are actually encountering.

Context definitions in this approach span the preprocessing and feature-extraction phases as well as the hidden Markov model. The preprocessing and feature-extraction phases are encoded at design time and are not adjusted at run time. They may introduce inaccuracy in context definitions that learning cannot address. A single microphone sensor is used to detect the context in this approach and the issues of sensor unavailability and sensor unsuitability are not addressed. In this approach the authors are only concerned with detecting the context using audio data, however if the approach was generalised to other sensors then the issue of sensor suitability would become an issue.

This approach depends on expert knowledge to encode the preprocessing and feature-extraction phases, and also to correctly label the training data. However the developer or user does not have to explicitly define or adjust the entire context definition therefore it is somewhat knowledge autonomous.

2.2.3.3 Structural learning

Albinali et al. present a framework for detecting human-activity contexts from sensor data (Albinali et al., 2007). The research is motivated by a number of applications, including observation of patient activity for early diagnosis of medical conditions. They apply a naïve Bayes classifier to identify contexts from sensor data.

In this framework training data is gathered and labeled with the correct context by the developer. Sensor data is preprocessed prior to being used for learning network probabilities. The first phase of preprocessing is *feature encoding*, where sensor data is discretised to a set of features based on predefined thresholds. The second phase of preprocessing is *feature selection*, where features for learning are selected. Features that do not occur in the training data, i.e., discrete states that are never active, are ignored. Then multicollinearity analysis, a statistical method for measuring correlation, is carried out to identify features that are redundant. If two or more features are highly correlated then only one needs to be used as an input to the naïve Bayes classifier.

Due to the uncertain set of features that remain after preprocessing the framework selects the network structure automatically. A hill-climbing algorithm is used to iteratively increase the number of nodes and links in a network. At each iteration supervised-parameter learning is used to learn probabilities for the current network using training data and the network's effectiveness for identifying known contexts is measured. The most effective network is selected for run-time interpretation of sensor data.

Analysis This approach applies supervised learning and a naïve Bayes classifier to learn context definitions. The accuracy of learned context definitions depends on training data being representative

of the run-time environment. This is illustrated by their approach of discarding features that do not occur in the training data, on the assumption that these features will not be useful in the run-time environment. Context definitions are not adjusted at run time.

Learned context definitions span the feature-encoding step as well as the naïve Bayes classifier probabilities. The feature-encoding step is fixed at design time and may introduce inaccuracy in learned context definitions that cannot be corrected by learning.

An interesting feature of this approach is the elimination of redundant sensor data using multi-colinearity analysis. This process goes some way towards addressing the sensor-unsuitability issue in that it evaluates the usefulness of sensor data. The current version eliminates redundant features (particular sensor data) rather than sensors, however it is reasonable to envision sensors being eliminated by extending the process. However a key point is that features are eliminated based on how redundant they are *when compared to other features.* The usefulness of features *to the application* is not evaluated during feature selection. Even highly-correlated features differ to a degree and this difference could capture important information for detecting the context, i.e., the small differences may be relevant to the application.

In contrast their structural-learning approach for selecting the network structure evaluates features based on how useful they are for detecting the context. This captures their usefulness to the application and if extended could capture the usefulness of sensors to the application. However the challenge of integrating new sources of sensor data at run time is not addressed and would be obstructed by the fixed feature-encoding step, which only knows how to discretise data forseen by the developer. The issue of sensor unavailability is not considered.

This approach depends on expert knowledge to define the feature-encoding step, as well as to label training data. As with other Bayesian network approaches the learning process gives this approach some knowledge autonomy.

2.2.3.4 Summary

Bayesian networks are the most frequently-applied approach to learned context definition. They require supervised training to learn network probabilities. This introduces a dependency on the developer or user to explicitly label sensor data with associated contexts, which is a major obstruction to knowledge autonomy. Context definitions that are accurate for the run-time environment require run-time training data, which requires that every application has a human in the loop. This goes against the vision of pervasive computing where devices disappear into the background, and may not be feasible for all context-aware applications.

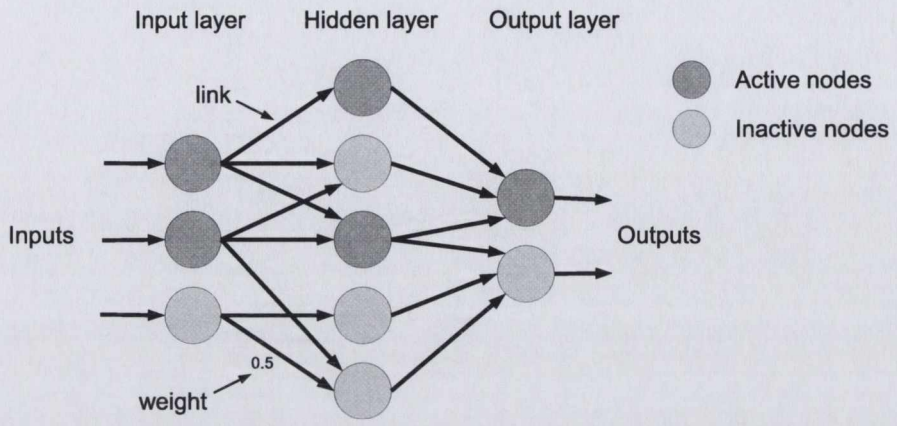

Figure 2.12: An artificial neural network

Other examples of learning approaches that apply Bayesian networks to context definition include Chang et al. (Chang et al., 2007), Harris and Cahill (Harris & Cahill, 2005a), Muhlenbrock et al. (Muhlenbrock et al., 2004) and Brdiczka et al. (Brdiczka et al., 2006).

2.2.4 Artificial neural networks

A number of approaches have applied artificial neural networks (ANNs) to learning context definitions. These approaches apply ANNs to learn patterns or clusters within sensor data that indicate contexts. ANNs are computational models that mimic how the brain processes information using networks of neurons (Russell & Norvig, 2003). Networks are composed of layers of nodes connected by directed links, with an input layer, a number of hidden layers, and an output layer (Fig. 2.12). ANNs reason by taking inputs in to their input layer and then spreading activation across the network to arrive at some conclusion (node activation) at the output layer.

Nodes in the network are activated depending on their inputs, which may come from other nodes or from outside the network. Each link between nodes has a numeric *weight* that defines the influence of the link. A node evaluates its activation status by calculating the weighted sum of its input links and passing this sum through an *activation function.* The activation function is designed to produce active outputs (~ 1) when the "right" inputs are provided and inactive outputs (~ 0) when the "wrong" inputs are provided. ANNs learn by adjusting the link weights that influence node activation. Fig. 2.12 shows an example of an ANN. The inputs cause two of the nodes in the input layer to activate and produce an output. Their outputs cause a further two nodes to activate in the hidden layer, which in turn cause a node to activate at the output layer. For a more detailed discussion of ANNs see (Russell & Norvig, 2003).

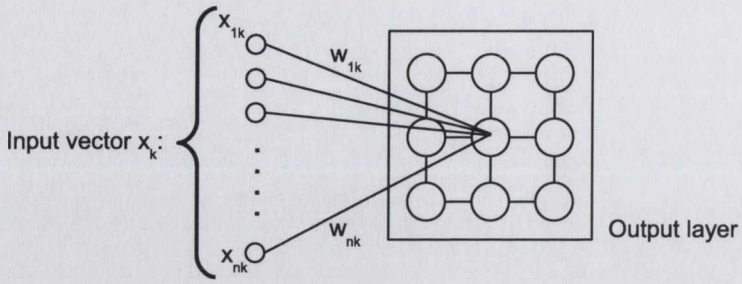

Figure 2.13: A self-organising map from (Van Laerhoven & Cakmakci, 2000)

2.2.4.1 Self-organising maps

Van Laerhoven and Cakmakci apply an ANN approach to learn user-activity contexts from sensor data (Van Laerhoven & Cakmakci, 2000; Van Laerhoven, 2001). They apply a self-organising map (Kohonen, 2001), a type of ANN, to learn context-definitions that are personalised to the particular user. Self-organising maps are neural networks with topologically-related nodes, usually organised as a 2-d grid. They process data to produce a low-dimensional representation of high-dimensional data.

In a self-organising map each node has a unique vector of weights, with one weight per input source. Self-organising maps learn by adjusting the weight vectors of nodes within the map. Numerical input data is combined in a vector form and mapped to a representative node (Fig. 2.13). The representative node is the node with the numerically closest weight vector to the input vector. After it is identified the representative node adapts its weight vector a little towards the input-vector values so that it better represents them. Over a number of iterations the weight vectors of nodes become representative of the weight vectors of frequently-occurring clusters of similar input data.

Fig. 2.14 shows a visualisation of nodes and data clusters from (Van Laerhoven & Cakmakci, 2000), where x and y are the self-organising map axes and the z axis is the number of node occurrences. It should be noted that in this figure the clusters are labeled with context names, however these labels were not learned. The self-organising map learns about patterns of input data and the result of learning is simply a set of unlabeled nodes, their weight vectors and their number of occurrences.

Van Laerhoven and Cakmakci apply a self-organising map to cluster sensor data from a wide variety of sensors including accelerometers, infrared sensors, carbon monoxide detectors, microphones, pressure sensors, temperature sensors, touch sensors and light sensors (Van Laerhoven & Cakmakci, 2000; Van Laerhoven, 2001). A *sensor* layer gathers data from these sensors and passes it to a *cue* layer. The cue layer preprocesses sensor data to a set of cues. Cues must be numeric to facilitate their comparison to weight vectors in the self-organising map, therefore the cue layer uses functions

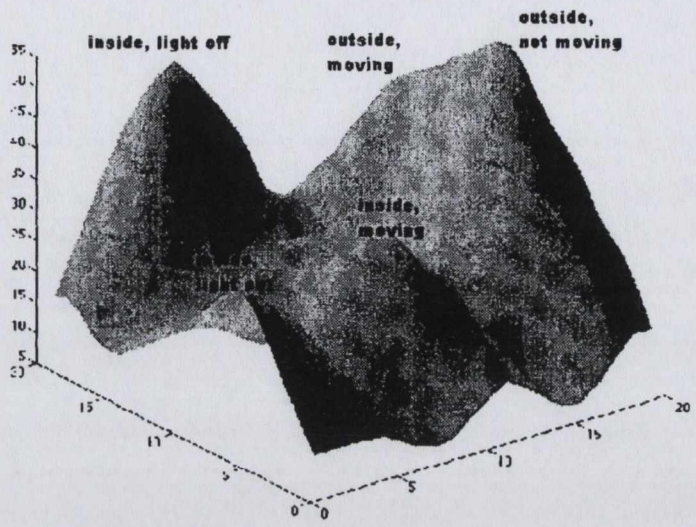

Figure 2.14: Clusters of data in a self-organising map from (Van Laerhoven & Cakmakci, 2000)

like averaging, standard deviation and fast Fourier transformation to process data. A *clustering* layer takes features as input, combines them in an input vector and passes them to a self-organising map, which learns a set of unlabeled clusters as described above. Finally clusters are labeled by the user in a *supervision* layer, in a process called *supervised classification*. These processes combine to learn context definitions at runtime.

Analysis This approach applies unsupervised learning of a self-organising map to learn context definitions. The initial clustering of data is unsupervised as sensor data is not labeled with the context, however learned contexts are not directly useful to the application. A subsequent supervision layer is required to define the relevance of clusters to the context-aware application.

Context definitions in this approach span the cue layer, the clustering layer and the supervision layer. The cue layer processes are fixed at run time but do not discretise sensor data to discrete states as in other approaches. Instead they apply statistical methods to summarise data. However there may still be some information loss during processing, e.g., statistical outliers are lost with averaging and standard deviation. Therefore the cue layer may introduce inaccuracy in learned context definitions that cannot be corrected at run time. The issues of sensor unavailability and unsuitability are not considered.

This approach depends on expert knowledge to define the cue layer processes and also to label contexts so that they are useful to the application. The clustering process gives this approach limited knowledge autonomy.

2.2.4.2 SCM

The Symbol-string Clustering Map (SCM) (Flanagan, 2005a,b) is a product of the Nokia Research Centre that arose out of research into context-sensitive user interfaces for mobile devices. The goal is to tailor the menu options presented to the user depending on their context. It addresses the limitation of self-organising maps that can only cluster numeric data.

In the SCM there are a set of nodes, each of which has a unique *symbol string*. Each node has a weight vector similar to the self-organising map, and each weight in its vector is associated with a particular symbol in the node's symbol string. Sensor input to the SCM is also in the form of a symbol string. The representative node for an input string is identified by comparing the symbols in the input string with those in each node's symbol string. The weights of overlapping symbols are summed for each node, and the node with the highest overall weight (the greatest overlap) is selected. The representative node's weight vector is updated by increasing the weights of overlapping symbols slightly. Clusters are identified based on the nodes that receive the highest numbers of updates, in the same way as for self-organising maps.

In this approach sensors for acceleration, atmospheric pressure, temperature, humidity, sound and location are used. Sensor data and user actions are recorded as training data for unsupervised and supervised learning. The recorded user actions label the data. Sensor data is discretised to a set of symbols in a preprocessing step and then processed by the SCM to identify clusters. Clusters are then associated with actions in a supervised-learning step. The training data is processed again and actions are probabilistically associated with clusters by calculating the ratio of cluster occurrences when the user took a particular action to the cluster occurrences when they did not. These probabilities are used to customise the mobile-device menu depending on the context.

Analysis This approach applies unsupervised learning and SCM to learn context definitions. The initial clustering of data is unsupervised, however the application cannot use the learned clusters directly. The supervised-training step probabilistically associates actions with clusters to complete the context definition.

Context definitions in this approach span preprocessing to symbols, the SCM and the action probabilities. The preprocessing step is fixed at design time therefore it may introduce inaccuracy in context definitions that cannot be corrected at run time. The issues of sensor unavailability and unsuitability are not addressed.

Training data for the supervised-learning step is implicitly labeled by the user when they take an action, therefore this is not a knowledge-intensive process. However, the process for discretising

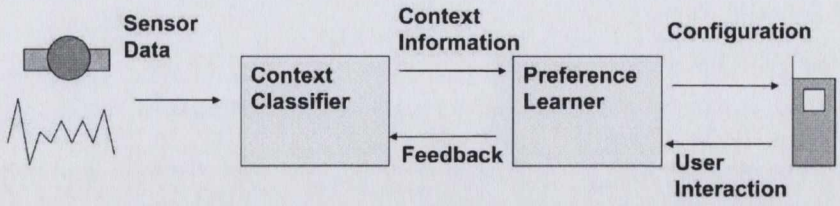

Figure 2.15: SenSay two-step approach from (Krause et al., 2006)

sensor data to symbols depends on expert knowledge therefore this learning approach is not completely knowledge autonomous.

2.2.4.3 SenSay

SenSay is a context-aware mobile device developed at Carnegie Mellon University (Krause et al., 2006). The goal of the research is to make the mobile device learn user contexts and automatically modify its settings based on these contexts. Sensor input is provided by various wearable sensors including accelerometers, thermometers, galvanic skin response sensors and microphones. The authors describe a two-phase approach for learning context definitions (Fig. 2.15). The *context-classifier* phase learns how sensor data is clustered to define contexts and the *preference-learner* phase learns how user preferences are influenced by those contexts.

The context-classifier preprocesses sensor data using running averages, sums of absolute difference and fast Fourier transformations. These processes must produce numeric outputs as they are used as inputs to a self-organising map. The self-organising map identifies clusters of sensor data as in the approach described in Section 2.2.4.1. The preference learner then uses a Bayesian network to learn the probabilities of user preferences given contexts. Since the set of contexts is unknown the network structure must be learned, and they apply a K2 structure-learning algorithm (Cooper & Herskovits, 1992) which iteratively builds networks to identify the best combination of nodes for predicting user preferences.

Initial learning in this approach is based on prerecorded sensor data and is carried out before runtime. Sensor data is recorded in contexts and labeled with user preferences by the developer. The self-organising map is trained to identify clusters using the sensor data, and the Bayesian network learns how user preferences are predicted using the labeled training data. The approach also facilitates learning at run time by employing a buffer to record recent sensor data and user preferences. The self-organising map and Bayesian network are then retrained at run time.

Analysis This approach applies unsupervised learning with a self-organising map and supervised learning with a Bayesian network to learn context definitions. Training data is recorded at run time so that learned context definitions are relevant for the run-time environment.

Context definitions in this approach span preprocessing, the self-organising map and the Bayesian network. As in the other self-organising map approach the preprocessing step does not discretise sensor data to discrete states but the statistical methods it uses may cause information loss. This process is fixed at design-time and may introduce inaccuracy in context definitions that cannot be corrected at run time.

As in the Bayesian network approach proposed by Albinali et al. (Section 2.2.3.3) this approach carries out structural learning of the Bayesian-network structure. In the course of structural learning the usefulness of clusters for predicting user preferences is evaluated, which suggests that this approach evaluates sensor suitability. However clusters represent multi-dimensional data from many sensors and the usefulness of particular dimensions (sensors) cannot be distinguished. As was the case for Albinali et al. the challenge of integrating new sensors at run time is not addressed and would similarly be obstructed by the preprocessing step, which only knows which statistical methods to apply to sensors forseen by the developer. The sensor unavailability issue is not addressed.

Similar to the SCM-based approach training data for the supervised-learning step is implicitly labeled by the user so gathering training data is not a knowledge-intensive process. However the preprocessing step is based on expert knowledge therefore this learning approach is not fully knowledge autonomous.

2.2.4.4 Summary

These approaches to learning context definitions apply artificial neural networks to learn clusters of sensor data that are indicative of the context. The learning process outputs clusters that have unknown relevance to applications as they are unlabeled, and a supervised step is necessary to define how clusters affect the application.

These clustering approaches are best described as flexible elements of the discretisation process. A potential issue is that clusters emphasise sensor data that occur frequently, and infrequently occurring data may go unnoticed. The frequency of occurrence does not necessarily correlate to their importance to an application for identifying the context. Sensor data that rarely occurs may be equally or more important to the application for this purpose. As with other discretisation approaches it is important that this useful information is not lost during discretisation (Section 2.2.2).

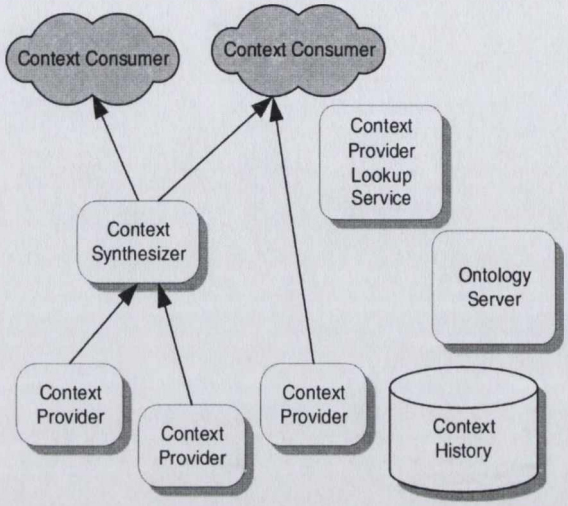

Figure 2.16: Gaia context infrastructure from (Ranganathan & Campbell, 2003)

2.2.5 Reinforcement learning

This section describes approaches that apply reinforcement-learning techniques to learn context definitions. Reinforcement learning is a computational approach to learning from interaction with the environment (Sutton & Barto, 1998). These approaches depend on *feedback* from the environment to learn the utility of actions in different situations. Feedback is interpreted by an application-specific *reward model* that calculates how effective the action was and produces a representative numeric reward. Reinforcement learning identifies the best action to take in different situations by comparing the accumulated rewards for each possible action, i.e., the action's *utility*. In context-aware systems reinforcement learning is applied to learn how applications should select actions in contexts.

2.2.5.1 Gaia

The Gaia project carried out at the University of Illinois provides a middleware for building context-aware applications (Ranganathan & Campbell, 2003). They apply reinforcement-learning to learn appropriate application behaviour in different contexts. The middleware includes a context infrastructure for context-aware applications (Fig. 2.16). *Context providers* represent sensors or other sources of context information. They may reason about and process raw sensor data to infer low-level context, using logic structures such as fuzzy logic and first order logic. *Context synthesizers* identify high-level contexts by combining data from context providers. *Context consumers* are the context-aware applications that interpret context and adapt their behaviour. The *context-provider lookup*

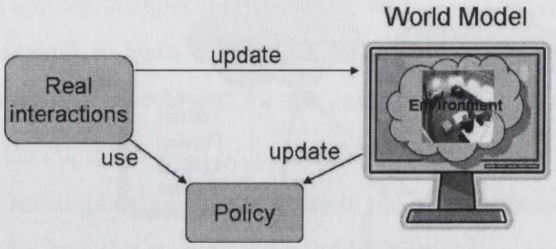

Figure 2.17: Zaidenberg et al's learning and interaction approach from(Zaidenberg et al., 2009)

service facilitates lookups for context providers and synthesizers at run time.

In the Gaia middleware applications may explicitly encode how context is interpreted using rules, or learn how context affects them using various algorithms including reinforcement learning, although the specific algorithm is unspecified. The set of contexts relevant to the application are specified by the developer at design time. The application learns at run time by taking actions in contexts and observing the user's reaction (feedback). Depending on the user's reaction the application increases or decreases the probability of taking that action again in the same context.

Analysis This approach applies a reinforcement-learning algorithm to learn context definitions. Context definitions span the context providers and context synthesisers that preprocess sensor data as well as the learned action-probabilities in the context consumer. Only the action probabilities are flexible at run time, and the preprocessing steps may introduce inaccuracy in the context definition that cannot be corrected at run time. Gaia provides a lookup service for finding context providers and synthesizers at run time, therefore this approach has the potential to handle the failure of particular sensors as long as at least one context provider of each required type is available. The issue of sensor unsuitability is not considered.

This approach assumes that a user is available to provide feedback to the application. Such a requirement is very similar to that of supervised learning where expert knowledge is used to explicitly label training data. In this case the user implicitly labels the context with an action by using their expert knowledge to provide explicit feedback about actions. The context providers and synthesizers of each application also depend on expert knowledge to define how they process data, therefore this approach has limited knowledge autonomy.

2.2.5.2 Zaidenberg et al.

Zaidenberg et al. learn context definitions for a context-aware personal assistant using reinforcement learning (Zaidenberg et al., 2009). Their approach learns to associate actions with perceived user situations based on feedback provided by the user in reaction to actions. One of their goals is to accelerate the learning process. To achieve this they construct a world model in which many actions can be emulated without affecting the user.

Fig. 2.17 shows an overview of their approach. When the application interacts with a user it uses its current *policy*, a mapping from contexts to actions, to select actions. A fixed set of contexts are defined as predicates, e.g., *absent(user)* indicates the user is absent from their office. The developer defines a default policy for these contexts at design time so that initial actions are acceptable to the user. The approach then learns new policies at run time to customise context definitions for the user.

In this approach the reinforcement-learning technique does not interact directly with the environment. Instead the user's reactions to actions are recorded and used to update a *world model*. The world model has two parts– a transition model and a reward model. The transition model defines how actions cause transitions between contexts and the reward model defines the rewards for actions taken in contexts. These models are learned using supervised learning, based on the recorded user interactions. The approach then uses the world model to emulate the real world. The DYNA-Q reinforcement-learning algorithm (Sutton, 1990) is used to learn a new policy by interacting with the world model. The new policy replaces the existing policy and the data gathering process begins again.

Analysis This approach applies a reinforcement-learning technique to learn context definitions. As in the previous approach it assumes the existence of a human in the loop who provides feedback for actions. A potential issue with their approach is the completeness of the world model. When the approach is gathering training data it only executes the actions in the current policy, i.e., one action per context. The world model they learn only has knowledge of the transitions and rewards associated with this limited set of actions, therefore the new policy that is learned by the reinforcement-learning algorithm is similarly limited. The proposed approach is missing a significant element of reinforcement learning– *exploration*. Exploration is where actions other than the perceived best action are taken in order to learn new knowledge.

In this approach context definitions span the low-level predicates and the learned policy, and are also influenced by the world model as it dictates how the policy is learned. The set of predicates is defined at design time and may introduce inaccuracy in context definitions that cannot be corrected at run time. There is no discussion of the sensors that populate predicates with information so the

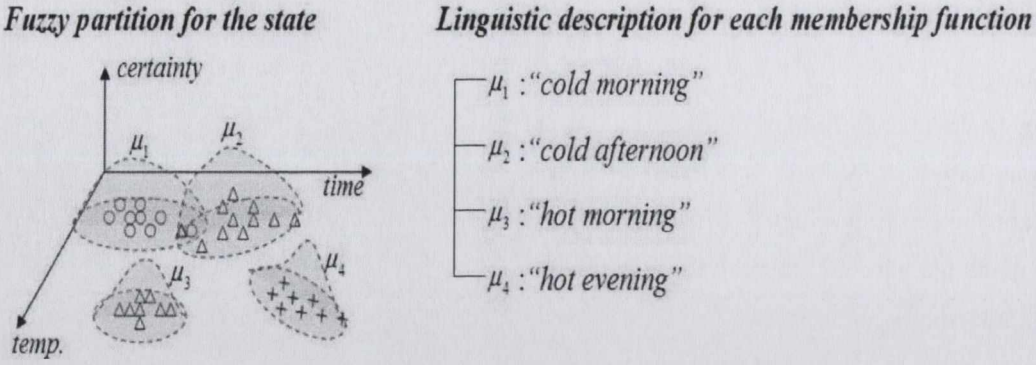

Figure 2.18: Fuzzy discretisation from (Ali et al., 2008)

issues of sensor unavailability and sensor unsuitability are not addressed. The approach depends on expert knowledge to define the predicates and their relationship to sensor data, and there is also a dependency on a user explicitly providing feedback based on their expert knowledge. Therefore this approach's knowledge autonomy is limited.

2.2.5.3 Fuzzy-state learning

Ali et al. (Ali et al., 2008) investigate how context-awareness is applied to assistive living for the elderly or disabled. The approach is applied to a scenario where a quadriplegic requires assistance when drinking coffee, eating a meal, taking a bath and using a computer. The context-aware application assists these activities by automatically turning on the TV, adjusting seat height, heating water etc.

They apply a reinforcement-learning technique to learn when it is appropriate to take these actions. Of particular interest is their approach to the discretisation of sensor data. Rather than define one-to-one mappings between sensor data and discrete states this approach uses fuzzy sets (Zadeh, 1965) to define these relationships. Each fuzzy set has a *membership function* that calculates the membership (0–1) of sensor data in the set. The membership represents the degree to which sensor data is *in* the set, i.e., a membership of 1 implies complete membership and 0 implies no membership. In this approach the developer defines *fuzzy states* to represent sensor data (Fig. 2.18). Each state has a membership function and a symbolic description, e.g., cold morning. By evaluating sensor data against the membership function of each fuzzy state its membership is established, and the membership represents the certainty that the application is in that context. For example, (12°C, 11:37am) might have a membership of 0.5 in the "cold morning" fuzzy state, whereas (10°C, 9:14am) might have a membership of 0.8 in the same fuzzy state. In the second case it is both more "cold" and more "morning". The developer captures this relationship between sensor data and membership

in the membership function, i.e., what it means to be a "cold morning". Sensor data may even have membership in multiple fuzzy states e.g. (12°C, 11:37am) could have membership in "cold afternoon" too.

In this approach the applied reinforcement-learning algorithm is Q-learning (Sutton & Barto, 1998). The fuzzy-state discretisation layer affects how the algorithm is applied when selecting and updating actions. When *selecting* the next action to execute in reinforcement learning, the possible actions in a state are compared based on their utility. However in this approach data may have partial membership in a number of states, each with its own set of action utilities. These are combined by scaling each state's utilities by the membership in that state, summing the scaled values for each action, and selecting the action with the highest overall utility. A similar technique is applied when *updating* action utilities in states. The reward for an action is applied to each active state, but is scaled by the membership (of the sensor values when the action was executed) in that state. This approach uses recorded sensor data that is labeled by the developer with appropriate actions, and performs learning prior to run time.

Analysis This approach applies reinforcement learning to learn context definitions. It is noteworthy for its discretisation process which uses fuzzy sets. The membership of data in sets represents the certainty that the application is in a particular context, and affects how actions are selected and updated.

Context definitions span the fuzzy-state discretisation and the learned knowledge of actions. Although the discretisation process does not define a one-to-one mapping to discrete states the relationship between sensor data and discrete states is still fixed. The membership functions defined by the developer at design time control this relationship and may introduce inaccuracy that cannot be corrected at run time. The issues of sensor unavailability and unsuitability are not addressed. This approach depends on expert knowledge to define the membership functions and also to label recorded sensor data, therefore it has limited knowledge autonomy.

2.2.5.4 Summary

In general reinforcement-learning techniques do not require labeled data however all of these approaches involve the developer or user for this purpose. This nullifies one of the advantages of reinforcement learning, i.e., it can learn using feedback *from the environment.* The main knowledge-intensive element of reinforcement learning should be the reward model that captures how useful actions are. It must encapsulate knowledge of the application goal and environment to achieve this.

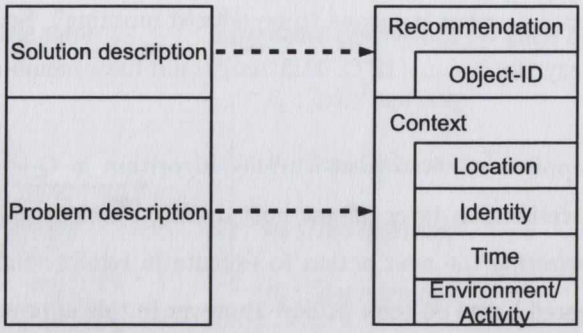

Figure 2.19: LISTEN case representation from (Zimmermann, 2003)

These approaches do not discuss the reward model as the feedback they receive already encapsulates how useful actions are.

2.2.6 Case-based reasoning

This section evaluates approaches that apply case-based reasoning (CBR) to learning context definitions. CBR (Kolodner, 1993) is a process for solving new problems based on known solutions to similar problems (cases). Cases consist of problem-solution pairs. A CBR application goes through four steps when it solves a problem (Aamodt & Plaza, 1994): *retrieve* the most similar case based on a *similarity function*, *reuse* the knowledge in that case to solve the new problem, *revise* the proposed solution by evaluating its success and repairing any faults, and *retain* new knowledge as a new learned case or by modifying existing cases. Approaches that apply CBR to learning context definitions store context data and application actions as cases.

2.2.6.1 LISTEN

A framework for context-awareness based on CBR was developed as part of the LISTEN project (Zimmermann, 2003). The goal of the project is to address context-awareness in audio-augmented environments and provide users with personalised audio experiences, e.g., in museums and galleries. The framework for context-awareness has four main layers. A *sensor layer* collects sensor data from sensors in the environment. A *semantic layer* maps sensor data to an application-specific context model. This is a multi-step process: sensor-data is discretised to entities in the context model, entities are related to each other to infer high-level contexts (using association, aggregation and composition), and the context is compared to previous contexts to observe its progression over time. A *control layer* generates commands to execute and passes these to an *indicator/actuator layer* that executes them.

The LISTEN framework applies CBR by mapping the problem-solution pairs of a case description

to context-recommendation pairs (Fig. 2.19) in the control layer. Four elements of context within a case (location, identity, time and environment/activity) are populated by different sets of entities in the context model. The combinations of these entities represent a four-dimensional vector of contexts. At run-time the current case is identified by populating it using the currently-active context-model entities. This case is compared to stored cases based on its location in the four-dimensional vector, with a high weight on the location, time and identity of the user. The solution of the nearest stored case is passed to the indicator/actuator layer for execution.

Analysis Context definitions span the semantic layer where sensor data is discretised and the control layer where CBR is applied to identify actions to take. Sensor unavailability and unsuitability issues are not considered.

Crucially this approach is missing the third and fourth steps of CBR described above, namely, it does not revise case solutions and retain new knowledge. This approach doesn't learn knowledge beyond that encoded by the developer therefore it is a purely knowledge-intensive approach. We include it as an introduction to the next approach, MyCampus (Sadeh et al., 2005), which does not give a complete description of how cases are related to context or how similar cases are selected.

2.2.6.2 MyCampus

The MyCampus project (Sadeh et al., 2005) conducted at Carnegie Mellon University investigates and experiments with pervasive computing technologies. The project focuses on a semantic web infrastructure for re-usable, context-aware services with an emphasis on privacy and personalisation. They apply CBR to address usability issues associated with capturing user preferences for personalisation of context definitions.

In the MyCampus infrastructure sources of context data such as sensors are represented as semantic web services. A context-aware application operating on behalf of a particular user can access these sources to identify the user's context. The infrastructure supports yellow-page lookups for resources at run time.

A central element of the infrastructure is an *e-wallet* that stores each user's resources such as their personal characteristics, schedule and preferences. Preferences may refer to any context-aware application and this creates a challenge for preference editing. Developing a unique preference-editing interface per application is prohibitive, and general purpose editors are too complex for normal users. To address this issue they apply CBR to learn user preferences. This approach is applied in a single application– context-aware message filtering.

The message-filtering application chooses when to notify the user of received messages based on the message and recipient's context. Initially the application gathers a set of sample cases by delivering messages and observing user feedback, i.e., the appropriate action to take. The application stores the context and the appropriate action for each message as a new case. The context data they store in each case is the message type and sender, and the recipient's location and activity. When sufficient cases are stored the application uses these cases to select delivery options for new messages, although the exact similarity function they use for comparing cases is not described.

Analysis The MyCampus project gives a brief description of how CBR is applied to learn personalised context definitions. Importantly they describe how new cases are learned using user feedback, so this CBR approach has the potential to learn new context definitions. Sensor data is recorded in cases along with user actions, therefore this approach is best classified as a supervised-learning approach as it depends on example inputs and outputs from which to generalise.

The sensor data used in the example application is from high-level sources, e.g., the user's calendar provides their location and activity, and the message provides the message type and sender. No low-level sensors are described, however other CBR approaches require discretisation of low-level sensor data (Kofod-Petersen & Aamodt, 2006; Ma et al., 2005; Zimmermann, 2003) and this is most likely the case for the MyCampus approach also. If discretisation is fixed at run-time (as is the case in other approaches) then it may introduce inaccuracy in context definitions that cannot be corrected.

The MyCampus infrastructure supports yellow-page lookups of resources at run time therefore it can handle the unavailability of individual sensor instances, however it requires that at least one instance of each required type be available. The issue of sensor suitability is not addressed.

The approach requires expert knowledge to define the similarity metric for comparing cases, as well as to define any discretisation process. The user is also needed to implicitly label sensor data with actions so that cases can be constructed. Therefore the knowledge autonomy of this approach is limited.

2.2.6.3 Summary

These projects apply CBR to context-awareness. The MyCampus project demonstrates that CBR-based approaches can learn new cases and therefore new context definitions at run time. A key element for successfully applying CBR is the similarity function that is used to compare cases. This function dictates how newly-encountered cases relate to cases for which there are known solutions. In context-aware applications the similarity function compares cases based on sensor data. It is defined

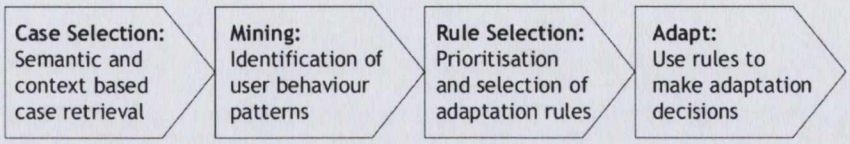

Figure 2.20: Data mining of contexts from (Tsang & Clarke, 2007)

by the developer at design time and therefore only compares cases based on the sensors and data that the developer forsees. This may prevent new sensors being incorporated into the application at run time.

Other approaches including AmbieSense (Kofod-Petersen & Aamodt, 2006) and Ma et al. (Ma et al., 2005) have applied CBR to context-awareness, however similar to the approach used in the LISTEN framework they do not learn new cases so they are knowledge-intensive approaches.

2.2.7 Data mining

This section describes approaches that apply data mining to learn context definitions. Data mining is the process of identifying relevant and useful patterns and relationships within large amounts of data, which can subsequently be used to make predictions (Callan, 2003). Individual records within the data are referred to as *cases* and the collection of records to be mined is referred to as the *case set.*

2.2.7.1 User-preference mining

Tsang and Clarke (Tsang & Clarke, 2007) apply data mining to the personalisation of user context definitions. The work is motivated by the difficulty of manually specifying user preferences in existing approaches. The premise of their solution is that users are habitual and their habits are hidden within their historical data.

Their approach gathers a case set by recording user behaviour and sensor data over time. A single case consists of a number of different elements. For example, a case for a restaurant application might be "{preference=chinese, cuisine=chinese, cost=cheap, meal=lunch, day=wednesday}" (Tsang & Clarke, 2007). The *preference* variable refers to the user's cuisine preference, *cuisine* and *cost* variables refer to properties of the restaurant, *meal* and *day* refer to context information. Possible case-element values are limited to those defined in a *user model.*

Fig. 2.20 shows an overview of their four-stage approach for mining context definitions. The *case-selection* stage selects a set of relevant cases from the overall case set. Relevant cases fulfill two criteria–

they are semantically similar to the current problem being solved and they have similar context. Semantic similarity is defined by the object hierarchy in the user model, e.g., when recommending a restaurant relevant cases include those related to both restaurants and cafés, as they share a parent *dining* object. Context similarity is defined by matching values, e.g., if the current context is "dinner time on a weekend" then it only selects cases where the user made decisions in the same context. The *mining* stage applies an association-mining approach to identify relationships or correlations between items in the case set. The particular algorithm they use is the Apriori algorithm (Agrawal et al., 1996), which discovers associations by determining the frequency that case-element values occur together. The output of mining is a set of rules in the form of predicates, e.g., cost=cheap => restaurant-choice=chinese, where cheap and chinese are case-element values. Data mining may yield a large number of rules so the *rule-selection* stage evaluates which rules should be used. They employ a common practice from association mining which is to compare rules in terms of *confidence* and *support*. Confidence represents the accuracy of the rule, i.e., the proportion of cases that it describes correctly. Support represents the number of cases that contribute to the rule. These two characteristics are used to select the best set of rules to execute. The *adapt* stage then executes the selected rules.

Analysis This approach applies association mining to learn context definitions. A case set of user actions in different contexts is recorded and mined. The approach is best classified as a supervised-learning approach as it depends on correct inputs and outputs to generalise from. The process of gathering sensor data is not described, however the symbols described in cases are high-level concepts which may need to be derived, e.g., infer "cheap" from menu prices.

Context definitions span any preprocessing of recorded symbols in cases and the mined association rules. The preprocessing element may introduce context-definition inaccuracy that this approach cannot correct at run time. The issues of sensor unavailability and unsuitability are not considered.

This approach depends on expert knowledge to define any discretisation process and also to implicitly label sensor data with actions. These requirements limit its knowledge autonomy.

2.2.7.2 Summary

Data mining approaches share a number of parallels with case-based reasoning. They both depend on a historical set of cases that link contexts to actions. Both approaches match the current context to existing cases and identify the best action to take. Data mining does not require a similarity metric to compare cases however, and instead relies on more generic measures of similarity such as the overlap in elements between cases.

2.3 Chapter summary

This chapter reviews the current state of the art in context definition. Table 2.2 summarises the extent to which existing approaches can adjust their context definitions at run time, their dependency on expert knowledge, and their ability to address sensor unavailability and unsuitability issues. Our survey shows that the techniques applied in related projects provide limited support for these issues, and none of the issues is fully supported by any approach.

Approaches provide flexible context-definitions at run time by enabling the user to make manual changes or by learning context definitions based on run-time sensor data. In each case the degree to which context-definitions are flexible is limited by fixed underlying discretisation processes. These processes may introduce inaccuracy in context definitions that cannot be corrected using existing approaches.

Learning approaches to context definition exhibit some knowledge autonomy by learning context definitions beyond those explicitly encoded by the developer. The knowledge-autonomy of learning approaches is limited by their dependency on developer knowledge to define discretisation processes that interpret sensor data, as well as other learning-technique specific structures such as the reward model (reinforcement learning) and similarity metric (CBR). Approaches that apply supervised-learning also depend on expert knowledge to label training data.

A number of approaches address the issue of sensor unavailability by loosely coupling sensors and applications. These approaches provide some kind of lookup service to enable connection to sensor instances at run time. This approach only provides limited support for the issue of sensor unavailability as it assumes that at least one sensor of each required type will be available at run time.

A small number of approaches address sensor unsuitability to a degree by evaluating the usefulness of sensor data. These approaches do not consider entire sensors, nor do they facilitate integrating new sensors beyond those specified by the developer therefore they provide very limited support for the issue of sensor unsuitability.

Our own approach bears various similarities to these existing approaches. It belongs in the run-time, context-learning category of approaches as it seeks to learn about context beyond the knowledge encoded by the developer at design time. Obviously it shares the underlying learning technique used by the reinforcement-learning based approaches in Section 2.2.5, however it does not rely on a user to provide regular feedback that indicates the context, i.e., some form of training data. Instead we take a more traditional reinforcement-learning perspective where feedback comes from the environment rather than explicitly from a user.

The work by (Ali et al., 2008) described in Section 2.2.5.3 has a similar focus to our own approach

in terms of considering raw sensor data. However their approach, which discretises sensor data using fuzzy logic, relies on a set of membership functions that limit the flexibility of context definitions at run time. Our own approach to discretisation has no such fixed structures.

The approaches taken by (Albinali et al., 2007) and (Krause et al., 2006) to select appropriate network structures for Bayesian networks bears a strong resemblance to our own approach to the search for the most suitable sensors. They evaluate the usefulness of particular pieces of sensor data from these sensors. However these approaches operate with fixed sets of sensors and fixed discretisation layers that limit which sensors can be used at run time. Our approach addresses this issue by learning how to discretise sensor data at run time for unforeseen sensors.

	Flexible context definitions at run time	Knowledge autonomy	Addresses sensor unavailability	Addresses sensor unsuitability
Knowledge-intensive design time				
Context Toolkit	–	–	○	–
Java Context-Awareness Framework	–	–	○	–
Sentient Object Model	–	–	○	–
Knowledge-intensive run time				
PACE	○	–	○	–
Context Studio	○	–	–	–
Bayesian networks				
Korpipaa et al.	–	○	–	–
Smith et al.	○	○	–	–
Albinali et al.	–	○	–	○
Artificial neural networks				
Van Laerhoven and Cakmakci	○	○	–	–
SCM	○	○	–	–
SenSay	○	○	–	○
Reinforcement learning				
Gaia	○	○	○	–
Zaidenberg et al.	○	○	–	–
Ali et al.	–	○	–	–
Case-based reasoning				
LISTEN	–	–	–	–
MyCampus	○	○	○	–
Data mining				
Tsang and Clarke	○	○	–	–

● = supported, ○ = limited support, – = no support

Table 2.2: Summary of state of the art approaches to context definition

Chapter 3

Design

Our analysis of state-of-the-art approaches to context definition shows that existing approaches provide limited support for adjusting context definitions at run time. This affects their ability to handle issues caused by uncertainty in the run-time environment– context misinterpretation, sensor unsuitability, and sensor unavailability. To address these issues context-aware applications should evaluate the accuracy of their context definitions and the suitability of sensors available at run time. State-of-the-art approaches to context definition are limited by their dependence on expert knowledge to provide either the entire context definitions of an application or elements of context definitions. These knowledge structures are not flexible at run time, so they limit an application's ability to evaluate and correct its context definitions. The structures also limit the types of sensors that an application can use to identify the context, so existing approaches only partly address sensor unavailability and do not address sensor unsuitability for identifying contexts. To overcome these limitations an approach must remove expert knowledge from its context definitions, i.e., become knowledge autonomous.

The research question that this thesis addresses is that of what techniques and algorithms are necessary to support accurate run-time context definition in unpredictable operating environments, including identifying suitable sensors from those available and accurately identifying the context from their sensor data. This thesis answers the research question by describing a novel process for learning context definitions at run time.

We begin by discussing existing definitions of context and their limitations for knowledge-autonomous context definition. We define the meaning of context from the perspective of the application. We then describe a representation for context definitions that is expressed on raw sensor data, to remove any dependency on expert knowledge. This representation also facilitates the use of a flexible set of sen-

sors. We then describe the KAFCA process that applies our theory of knowledge-autonomous context to learn context definitions. At the core of the process is reinforcement learning (Sutton & Barto, 1998), which we identify as the learning technique that is least dependent on expert knowledge. Reinforcement learning is used to learn mappings between sensor data and actions, which the application interprets to define context definitions. We define a set of algorithms around this learning technique that refine the discrete states on which learning is based to improve the accuracy of context definitions, and combine and evaluate sets of sensors to select the most suitable set from those available. These techniques and algorithms combine in a two-phase process that addresses the limitations of existing approaches by removing knowledge-intensive structures from context definitions. The remainder of this chapter provides a detailed description of the approach proposed in this thesis, and concludes with a summary of how the challenges identified in Section 1.3 are addressed.

3.1 Knowledge-autonomous context definitions

In order to design an approach where the application can interpret and adapt its context definitions autonomously we must first understand what context is. In this section we examine existing definitions of context and their usefulness for knowledge-autonomous context definitions.

We follow the description of the evolution of context-aware systems described in (Baldauf et al., 2007), from stand-alone applications to generic frameworks that support application development. Early research in context-awareness focused on defining context in terms of types of information. Schilit and Theimer describe context as location, identities of nearby people, objects and changes to those objects (Schilit & Theimer, 1994). Ryan et al. describe context as the user's location, environment, entity and time, and also people and objects in the user's environment (Ryan et al., 1998). Dey and Abowd's (Dey & Abowd, 2000) definition is widely cited:

> "any information that can be used to characterize the situation of entities (i.e., whether a person, place or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves."

More recent definitions classify context information along two dimensions: external and internal (Prekop & Burnett, 2003; Hofer et al., 2003). The *external* (physical) dimension refers to context that can be measured in the physical environment. The *internal* (logical) dimension refers to user context, e.g., goals, business processes, emotional state.

These definitions of context focus on context from a human perspective. They emphasise that context definitions should be based on internal representations of external entities, i.e., a world model

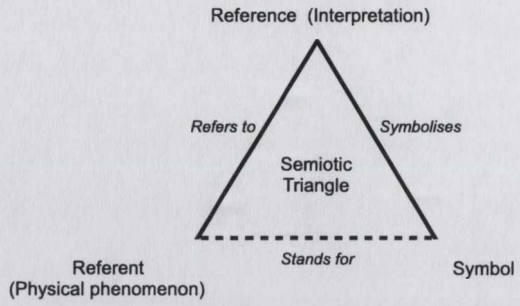

Figure 3.1: A semiotic triangle (adapted from (Ogden & Richards, 1923))

(Leahu et al., 2008). As such they are well suited to knowledge-intensive approaches as these approaches capture a developer's perspective of application contexts. However these definitions of context are not well suited to knowledge-autonomous approaches, as they are based on human-interpretable symbolism that the application cannot intuitively interpret, as it has no innate knowledge of locations, identities or objects.

The *semiotic triangle* (Ogden & Richards, 1923) can be used to explain why human symbolism limits an application's ability to introspect about its context definitions. The semiotic triangle is a model that explains how linguistic symbols are related to the real-world objects that they represent. The triangle combines three elements (symbol, referent and reference) in a reasoning model (Fig. 3.1). An internal symbol *stands for* an external referent, and the relationship between symbol and referent is captured by a *reference*. Nehaniv describes how the semiotic triangle defines meaning within software applications (Nehaniv, 1999). A physical phenomenon is internally represented by the application using a symbol. The mappings between the physical world and symbols are captured in an interpretation that signifies the meaning of the symbol to the application. The symbols alone are meaningless to the application without the interpretation. Context definitions based on human symbols must similarly have interpretations for those symbols provided so that the context definitions are meaningful to the application.

For example, the symbol *cold* is meaningless to an application as it has no innate understanding of *cold*. The symbol only becomes meaningful if the application somehow encapsulates an interpretation of *cold*, such as "*cold* is an undesirable context. Activate a heater to change the context to *warm*, which is a more desirable context". The application can only interpret human symbols as it has been instructed to interpret them. It cannot reason about the accuracy of the interpretations as it has no innate knowledge of the human reasoning they capture. Therefore these human-centric definitions of context do not facilitate application introspection about context.

A small number of existing definitions of context do not focus on human symbolism. For example Hull et al. describe context as aspects of the local environment (Hull et al., 1997), and Brown defines context as elements of the environment that the computer knows about (Brown, 1996). However these definitions are too vague to capture the effect of context on the application, or the relationship between sensor data and context.

For a context-aware application to be knowledge autonomous it must be capable of introspection over its context definitions. To achieve this we must define context in terms that are meaningful to the application and independent of human symbolism.

3.1.1 Introspective context

We require an application-oriented definition of context to enable introspection over context definitions. Gellersen et al. discuss the confusion that may arise from using the term "context" at different levels of abstraction (Gellersen et al., 2002). They describe three levels of abstraction– a specific occurrence of an aspect of the environment, e.g., a specific place; a particular aspect of the environment, e.g., location; and the entire environment that surrounds an application. We use the term "context" to refer to the entire real-world situation of an application.

To better understand the relationship between sensor data and context we begin by examining the motivation for context. A context-aware application perceives the environment through sensors that can produce thousands or even millions of unique values. For example a temperature sensor with range 0.00–100.00°C has 10,000 possible values. It is infeasible to encode how each value uniquely affects an application. Contexts refer to the environment at a much less fine granularity, therefore they have the effect of simplifying the application's perception of the environment. We therefore define a practical motivation for context:

> *Context simplifies an application's perception of the environment by generalising about sensor data.*

In knowledge-intensive approaches contexts are defined based on expert knowledge, e.g., values in the range 3–15°C indicate the context "cold". The sensor values in this range share some common relevance or meaning to the application (from the developer's perspective), which is symbolised by "cold". As discussed earlier the application has no innate knowledge of what "cold" is therefore it cannot introspect about its accuracy or relevance. Introspective context definitions must be based on meaning that is relevant to the application.

Although the meaning of context to the application has not been explored, a number of researchers

have investigated the meaning of sensor data. Nehaniv (Nehaniv, 1999) examines mathematical notions of meaning, stressing that the meaning of information is revealed in *how it influences* an agent. A discussion of sensor data in (Polani et al., 2001) suggests that meaning is grounded in *how it is used*, based on a notion originating in linguistics (Wittgenstein, 1968). Using this terminology we state that context-aware applications *use* sensor data to *influence* how they select the actions to be performed. Based on this usage we define meaning for context-aware applications:

> *The meaning of sensor data to a context-aware application defines how it affects action selection.*

As contexts are generalisations of sensor data they share the same meaning to the application as the sensor data they represent. We finalise our definition of context:

> *A context represents sensor data that causes the same action selection.*

Although this definition of context may appear overly generic it captures the meaning of context to an application and also the effect that the context has on its behaviour, i.e., how an application might reason philosophically about its context. In this definition context simply serves as a generalisation of stimulii that affect application behaviour. Further interpretation of context would involve human symbols and semantics that cannot be intuitively interpreted by an application. In the next section we consider how an application reasons more practically about its context by creating context definitions.

We assume that the set of actions available to the application are known a priori and therefore the set of application contexts are implicitly known. Although a human might define two different contexts, e.g., "at the cinema" and "in meeting", from the application's perspective they might be the same context– "set mobile phone to silent".

Unlike existing definitions of context this definition does not depend on human symbolism and instead focuses on context from the application's perspective. The definition is simple and abstract but it has a considerable implication for knowledge-autonomous context definitions. By comparing the meaning (influence on action selection) of different sensor data the application can evaluate if they belong in the same or different contexts. This comparison facilitates introspection about the accuracy of context definitions.

Furthermore, we will show that reinforcement learning can be applied to learn how sensor data influences action selection. As a result there is no requirement for the developer to define this meaning using expert knowledge. The use of reinforcement learning is discussed in detail in Section 3.4.

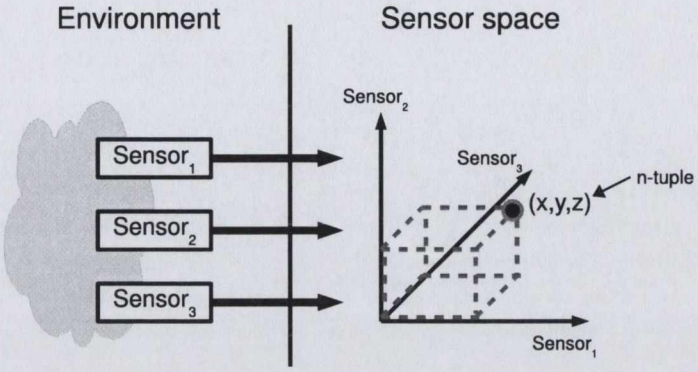

Figure 3.2: An application's perception of the environment

3.1.2 Representation of context definitions

Given this definition of context we now consider how context definitions should be represented. The representation must be flexible to support our requirements for knowledge autonomy (Section 1.3.3). The relationships between sensor data and contexts, and the set of sensors used to distinguish contexts must both be flexible. This facilitates adjusting context definitions at run time to address context misinterpretation and sensor unsuitability or unavailability.

3.1.2.1 Sensor-data representation

Context is interpreted from sensor data, so we begin by defining how sensor data is represented by the application. A context-aware application consumes sensor data from sensors in the physical environment. KAFCA uses the MoCoA middleware to interact with sensors so the complexity of communicating directly with sensors is abstracted away (Section 1.6).

To avoid knowledge-intensive structures that may limit the accuracy of context definitions (Section 2.2.2) the application must work with raw sensor data. The application infers its context from a set of n sensors. At a point in time the current readings from each sensor are combined as an *n-tuple*, and the n-tuple represents the application's current view of the environment. The set of all possible n-tuple combinations from n sensors lie within an *n-dimensional sensor space* (Figure 3.2). Each sensor's range of possible values represents a single dimension of the sensor space.

It should be noted that the application does not store the entire sensor space of tuples, rather the sensor space represents application awareness of the set of environmental situations it can distinguish. The set of possible tuples can be calculated by the application using sensor meta-data that describes their range of values and precision (Section 1.6). Any set of n sensors can be combined to form a

sensor space therefore this representation of sensor data addresses the requirement that the set of application sensors be flexible.

3.1.2.2 Context edges

In Section 3.1.1 we describe context as a generalisation of sensor data that cause the same action selection to occur. Based on the representation of sensor data we have just described contexts represent sets of similar n-tuples in the sensor space.

We make the assumption that sensor data is ordered. This makes the sensor space a *Euclidean space*, where the distance between points in the space can be measured. The ordered-data assumption is true for many sensors in the physical environment, e.g., distance, temperature, weight, height, voltage, pressure, speed, acceleration etc. Under this assumption it is our intuition that in general similar tuples will be co-located in the sensor space. This allows the application to define contexts as regions in the sensor space.

We exploit the co-location of similar tuples to define contexts as contiguous subspaces within the sensor space. Within a context all sensor tuples are homogenous, i.e., they share the same meaning. Each context is bounded by *context edges*. A context edge is a boundary to a region of sensor space where the meaning of tuples is homogenous. All tuples on one side of the context edge share the same meaning as they are within the context, while tuples on the other side have different meaning(s).

Contexts are defined in terms of their context edges. Rather than define explicit associations between tuples and their context they are implicitly associated by being located in the sensor space between the context edges of a context definition. The idea of defining contexts in terms of context edges is derived from a discussion of continuous state spaces in (Atkin & Cohen, 2000). Atkin argues that there are particular *critical points* in the space where decisions are made and that the continuous space between these points are irrelevant to the application. We equate critical points to context edges, as the application makes the decision to change its behaviour when its context changes, i.e., when the environment changes such that the current tuple is across a context edge from the previous tuple. Once it has adapted to its new context the exact state of the environment is irrelevant, that is, until another context edge is crossed.

Fig. 3.3 shows a number of tuples in a sensor space. Each tuple is colour-coded based on the action selection it causes. Context edges separate subspaces where the meaning of tuples is homogenous. The figure illustrates a sensor space where two different subspaces cause the same action selection, however they are separated by a third subspace that has different meaning. These similar subspaces are philosophically in the same context as they cause the same action selection, however in practice

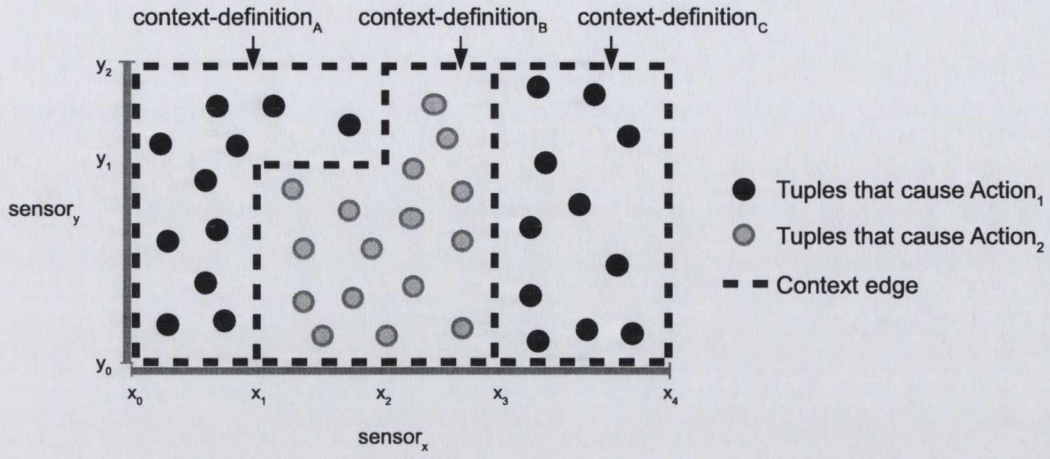

Figure 3.3: Context definitions in the sensor space

they each have their own context edges and therefore are defined in separate context definitions ($context - definition_A$ and $context - definition_C$).

It is important to distinguish between the concepts of context and context definitions. Contexts define how an application should behave in its environment, whereas context definitions are manifestations of application contexts in particular sensor spaces. The fact that a particular sensor space represents a context in two separate subspaces is a function of that sensor space. A different set of sensors and associated sensor space might represent the context in a single subspace or more than two subspaces, however this does not change the philosophy of the subspaces belonging to the same context.

Sensor spaces have previously been proposed as a means of representing context (Padovitz et al., 2005) but only using static, knowledge-intensive contexts expressed on the sensor space. With our approach the application can introspect about its context definitions by comparing the meaning of neighbouring tuples and adjust its context definitions appropriately. The context edges that it identifies at run time are accurate for the particular run-time environment.

It should be noted that although multiple context definitions can be expressed on the same sensor space not all contexts are necessarily identified by the same sensors in the sensor space. For example, in Fig. 3.3 $context - definition_A$ and $context - definition_B$ both have irregular shapes in the sensor space. In order to test if an application is in one of these contexts the values of both $sensor_x$ and $sensor_y$ must be considered, e.g., the application is in the context defined by $context - definition_A$ if the value from $sensor_x$ is between x_0 and x_1, or if the value from $sensor_x$ is between x_1 and x_2 and the value from $sensor_y$ is between y_1 and y_2. In contrast the application can test if the application is

in the context defined by $context - definition_C$ by considering only the value from $sensor_x$, i.e., the application is in the context defined by $context - definition_C$ if the value from $sensor_x$ is between x_3 and x_4. The value from $sensor_y$ does not affect whether the application is in this context or not.

This representation of context definitions addresses the issue of context misinterpretation at run time, as the relationship between sensor data and context is not fixed. Instead context definitions are defined in terms of context edges. The application can identify context edges at run time to define context definitions that are accurate for a particular run-time environment.

3.1.3 Critique

Our definition of context focuses on how context-aware applications react to their operating context by selecting actions. A limitation of this view of context is that it doesn't capture how sequences of values from sensors indicate a context. This limits the applicability of the approach to applications that simply react to changes in their operating environment. The challenge of knowledge-autonomously defining context definitions that capture how sequences of sensor data indicate the context is considered beyond the scope of this thesis.

Our definition of context and representation of context definitions emphasise context as a generalisation of sensor data. This is motivated by the wide range of sensor readings that sensors in the physical environment produce. We are particularly interested in such sensors as the physical environment is the user interface for context-aware applications, and key to transparent interaction with users (Section 1.2). Internal sources of data such as a user-profile, calendar, device status, or network status could be modeled as software sensors and integrated in context definitions, as long as their data was converted to numeric data that could be mapped to tuples in a sensor space. Such conversions would be sensor specific, and we do not consider this challenge to be within the scope of the thesis.

Our representation of context definitions also depends on ordered sensor data. Sensors that do not produce ordered data could also be integrated in context definitions, however it would not be possible to generalise about their sensor data based on regions in the sensor space. Each sensor value would have to be individually mapped to a context or generalised about based on expert knowledge. We do not consider the challenge of integrating such sensors within the scope of the thesis.

3.2 KAFCA

The KAFCA process applies techniques and algorithms to realise this theory of knowledge-autonomous context definition. Figure 3.4 shows a high-level overview of the elements of KAFCA. Interactions

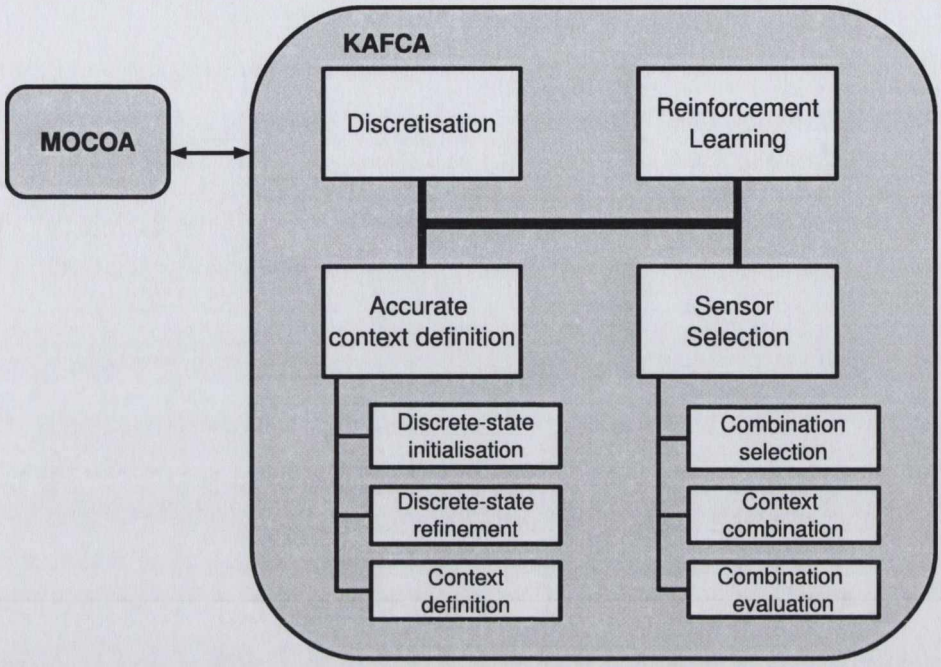

Figure 3.4: KAFCA processes

with the environment through sensors and actuators are delegated to the MoCoA middleware (Senart et al., 2006). This allows KAFCA to ignore low-level issues such as connections, communication protocols and routing, and focus on higher-level context-definition issues.

The KAFCA process is organised as a set of subprocesses that are executed to learn context definitions at run time. At the core of KAFCA are two subprocesses for accurate context definition and sensor selection. These subprocesses depend on the other two subprocesses within KAFCA– reinforcement learning and discretisation.

Reinforcement learning is a key enabler of the context-definition and sensor-selection processes as it is used to learn how sensor data influences action selection. This provides the meaning for sensor data that underpins our application-oriented definition of context. As discussed in Chapter 2 a variety of techniques have been applied in approaches that learn context definitions. Some degree of supervision is a common requirement among the surveyed approaches and this requirement introduces a dependency on expert knowledge. Supervision may take the form of data labeled with correct actions (Bayesian networks, CBR, data mining) or supervised labeling of learned contexts (artificial neural networks). Although supervision is a necessary requirement in most of these approaches it is not a necessary requirement for reinforcement-learning-based approaches. Reinforcement learning uses *feedback from the environment* to evaluate the effectiveness of actions. The surveyed reinforcement-

learning-based approaches in Chapter 2 simplify the problem of interpreting feedback by relying on a human in the loop to provide explicit feedback about actions, i.e., supervision. However feedback can come from sources other than users so there is potential for autonomous learning of context definitions. We therefore identified reinforcement learning as the most knowledge autonomous learning technique. The reinforcement-learning subprocess outputs a *policy* that maps sensor data (represented by discrete states or contexts) to actions, which is used by other KAFCA subprocesses to increase the accuracy of context definitions and identify suitable sensors.

As is the case for other approaches that apply learning techniques KAFCA must perform some *discretisation* of sensor data to make the learning task feasible. In related approaches discretisation encapsulates expert knowledge that is inflexible at run time, however KAFCA defines a flexible discretisation process based on *discrete states* that is independent of expert knowledge. The discretisation subprocess maps sensor data to discrete states that represent particular subspaces of the sensor space. The set of discrete states (and subspaces) is adjustable so the KAFCA process can adjust how sensor data is discretised. The discretisation subprocess also discretises sensor data to *contexts* as they similarly represent subspaces of the sensor space, although at a higher level of abstraction than discrete states.

The *accurate-context-definition* subprocess defines context definitions for individual sensors that interpret the context as accurately as is possible from that sensor's data. As discussed this sensor data must be discretised before reinforcement learning can be applied. In order to define context definitions that are accurate they must be expressed on discrete states that capture important changes in the environment (Section 2.2.2). The accurate-context-definition subprocess refines discrete states at run time to capture these important changes i.e. changes in application context. An initial set of discrete states is constructed for a sensor based on the meta data it provides. These discrete states are iteratively refined to improve their representation of context edges. When refinement is finished context edges are identified and context definitions are defined for a sensor.

The *sensor-selection* subprocess combines and evaluates different sets of available sensors to identify the best combination for detecting application contexts. Combinations are selected using an informed search, which uses the results of previous combination evaluations to guide its selection of future combinations. Each sensor in a combination has its own set of one-dimensional context definitions from the accurate-context-definition subprocess, and these definitions are combined to form n-dimensional definitions for a combination of n sensors. Application performance while using each combination is measured against a developer-defined metric, and the result feeds back into the search for better combinations. This subprocess continues until application performance is no long improved

by using different combinations of sensors.

Alg. 1 is a high-level algorithm that shows the sequence and iterations within the KAFCA process. The first phase of KAFCA iterates over the set of available sensors. Discrete states are initialised for each sensor, and then iteratively refined by learning a policy for the current set of discrete states and then splitting discrete states near context edges. After refinement a set of context definitions are defined for the sensor. The second phase of KAFCA iteratively selects the most suitable sensors for an application. In each iteration a new combination of sensors is selected, and their individual context definitions are combined. A policy is learned for these contexts and application performance is evaluated while selecting actions based on this policy.

As an example consider a simple application that is responsible for closing and opening office blinds. It discovers five sensors in its environment when deployed– temperature, light intensity, sound level, atmospheric pressure and humidity. Each sensor may contribute data that is useful for identifying the application context, however the application has no prior knowledge of these sensors.

In the first phase of KAFCA each sensor is considered in sequence. When it is under consideration a set of discrete states are initially defined for each sensor based on its meta data. These states are refined over a fixed number of iterations by learning how the application should behave in each discrete state and then updating discrete-state boundaries to reflect what has been learned. At the end of its individual consideration context definitions are derived for the sensor by identifying chains of similar, neighbouring states. In this example each of the five sensors will have their own context definitions at the end of phase one, that represent how their individual sensor data influences application behaviour. For example the temperature sensor might indicate that when temperatures drop below 11°C the shutters should close to conserve heat, the light intensity sensor might indicate that shutters should close above 60W to aid computer-screen visibility.

Once all sensors have been individually considered they can be combined in phase two of KAFCA. Sensors are combined by constructing n-dimensional rectangles from their contexts identified in phase one. For example, the temperature sensor has two context definitions from phase one– 0-11°C and 11-100°C, while the light-intensity sensor also has two context definitions– 0-60W and 60-500W. When combined these form four two-dimensional context definitions– (0-11,0-60), (11-100,0-60), (0-11,60-500), (11-100,60-500). The application is evaluated against a metric for its performance, and the result of this metric feeds back into the search for better combinations.

Algorithm 1: KAFCA Algorithm

```
/* PHASE 1: Accurate context definition */
foreach sensor s do
    Initialise discrete states using meta data for s;
    for i=1 to number of refinements do
        learn policy using reinforcement learning;
        refine discrete states;
    end
    define contexts for sensor s;
end
/* PHASE 2: Sensor selection */
while application-performance improves do
    select a new combination to evaluate;
    combine the context definitions of sensors;
    learn policy using reinforcement learning;
    evaluate performance using this combination;
end
```

3.3 Discretisation

The discretisation subprocess identifies discrete states and contexts to which raw sensor data belongs. Both discrete states and contexts represent subspaces within the sensor space. Raw sensor data is retrieved through the MoCoA middleware and combined in the n-tuple structure described in Section 3.1.2.1. An n-tuple is then discretised to a discrete state or context by comparing its coordinates in the sensor space to the subspaces they represent. The specific structure to which sensor data is discretised depends on the phase of KAFCA. During discrete-state refinement, for example, the process has not yet defined any contexts therefore sensor data is discretised to a discrete state. During sensor selection the contexts of each sensor will have been defined therefore sensor data is discretised to a context.

We first discuss how sensor data is discretised to discrete states. Our initial implementation of discretisation used a nearest-neighbour approach to define the subspace that is represented by a discrete state. Each discrete state had a centre point that marked the centre of its subspace. Discrete states were distributed around the sensor space, and sensor-data tuples were discretised to a discrete state based on their proximities to discrete-state centres. The distance between tuple coordinates and the centre points of discrete states was measured using the Euclidean-distance metric from coordinate geometry.

$$distance(a, b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + ... + (a_n - b_n)^2}$$

This simple discretisation process works well for one-dimensional spaces, however in spaces larger than one dimension it is difficult to reason about the boundaries between discrete states. In KAFCA

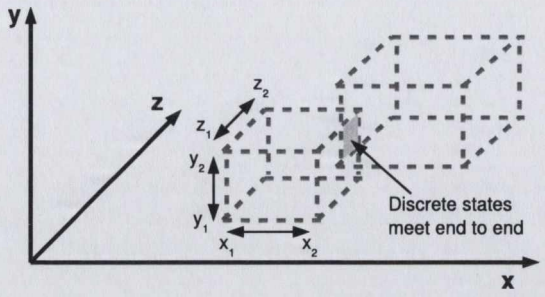

Figure 3.5: A discrete state in 3-dimensional space

reinforcement learning learns policies that map sensor data to actions at the granularity of discrete states, and context edges are identified at the boundaries between dissimilar discrete states. With a nearest-neighbour discretisation there is no explicit representation of discrete-state boundaries. The shape of each discrete state's subspace depends on its centre point and the center points of other discrete states. This made it too complex to define the location and shape of a boundary (and context edge).

Due to this complexity we changed our focus to a more regular discretisation based on n-dimensional rectangles. In this approach a discrete state represents a subspace in n dimensions. Its size in each dimension is defined by a *range* of values between a lower and upper bound. Fig. 3.5 shows the subspaces represented by two discrete states in three-dimensional space. Each discrete state's subspace is defined by its range of values in each dimension, and the boundaries of the subspaces are predictable.

Alg. 2 is an algorithm for identifying if discrete states are neighbours. Neighbours are identified by comparing their ranges in each dimension of their subspace. The number of dimensions in which their ranges overlap are counted. A dimension in which they do not overlap is also stored. If *n-1* of their n ranges overlap and the non-overlapping ranges of the states meet end-to-end (i.e. in range r the upper bound of state s_1 is the lower bound of state s_2, or vice versa), then their subspaces share a boundary and therefore they are neighbours. In Fig. 3.5 the ranges of the discrete states overlap in dimensions y and z, and the ranges in dimension x meet end-to-end.

The discretisation subprocess maps raw sensor data to a discrete state by comparing tuple coordinates to each discrete state's subspace. The algorithm for discretising sensor data to a discrete state is shown in Alg. 3. Sensor data is first combined in an n-tuple, and then compared to each discrete state. Each element of the n-tuple is compared to the range of a discrete state in the corresponding dimension. If all elements are between the lower and upper bounds of a discrete state then that state is identified as the representative of the sensor data and returned.

Algorithm 2: Neighbouring discrete-state identification algorithm

```
Input: Discrete states s1 and s2
Output: true if states share a boundary
numberOverlapping = 0;
nonOverlapping = -1;
/* compare each dimension */
foreach dimension dim do
    if dimension[dim] of s1 overlaps dimension[dim] of s2 then
        increment numberOverlapping;
    end
    else
        nonOverlapping = dim;
    end
end
/* check if their ranges overlap in n-1 dimensions and meet in the other */
if numberOverlapping equals (number of dimensions - 1)
and dimension[nonOverlapping] of s1 meets dimension[nonOverlapping] of s2 then
    states are neighbours;
end
else
    states are not neighbours;
end
```

Algorithm 3: Discrete-state discretisation algorithm

```
Input: Sensor data
Output: Representative discrete state
/* combine sensor values in an n-tuple */
foreach sensor s in n dimensions of current sensor space do
    add current reading from s to n-tuple;
end
/* find discrete state */
foreach discrete state d do
    foreach dimension dim of d do
        if tuple[dim] NOT within discrete-state range for dim then
            skip to next discrete state;
        end
    end
    if all tuple values within discrete-state ranges then
        return discrete state d;
    end
end
```

The discretisation subprocess also discretises sensor data to contexts. Contexts are defined on top of discrete states and therefore may take on irregular shapes. Rather than define the boundaries of a single context subspace we define contexts as collections of discrete states. The discrete states in a context represent contiguous regions of the subspace, i.e., they neighbour each other, and the

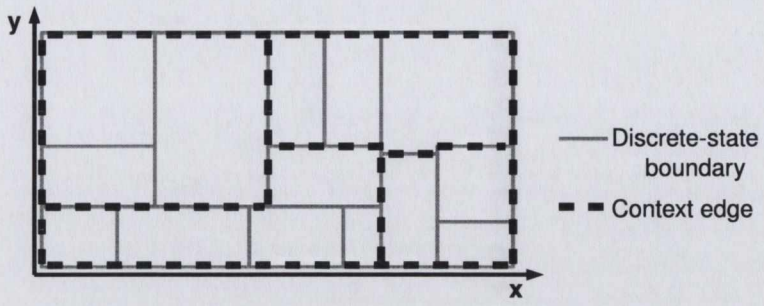

Figure 3.6: Contexts are collections of discrete states

discrete states at the edges of these contiguous regions define context edges (Fig. 3.6). Sensor data is discretised to a context by comparing tuple coordinates to the subspaces of discrete states within a context (Alg. 4). This algorithm compares tuple elements to discrete-state ranges in the same manner described in Alg. 3. However in this algorithm the context to which the representative discrete state belongs is returned rather than the discrete state itself.

Algorithm 4: Context-discretisation algorithm

```
Input: Sensor data
Output: Representative context
/* combine sensor values in an n-tuple */
foreach sensor s in n dimensions of current sensor space do
 | add current reading from s to n-tuple;
end
/* find context */
foreach context c do
 | foreach discrete state d in context c do
 | | foreach dimension dim of d do
 | | | if tuple[dim] NOT within discrete-state range for dim then
 | | | | skip to next discrete state;
 | | | end
 | | end
 | | if all tuple values within discrete-state ranges then
 | | | return context c;
 | | end
 | end
end
```

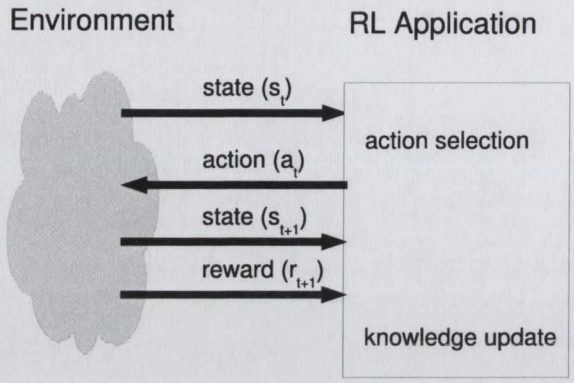

Figure 3.7: Reinforcement-learning steps

3.4 Reinforcement learning

The reinforcement-learning subprocess takes a set of discrete states or contexts as input and outputs a policy that maps the input states to application actions. Learned policies are used across the KAFCA process, to identify discrete states that should be refined, to define contexts, and to select actions during evaluation of different sets of sensors.

Reinforcement learning learns the effectiveness of actions in states through trial and error (Callan, 2003). In KAFCA a state may be either a discrete state or a context depending on the phase of the process. The reinforcement-learning subprocess has five steps that are repeated until a policy is learned (Fig. 3.7):

1. The current *state* s_t is observed at time t.
2. An *action* a_t is selected and then executed.
3. The resultant transition to a (possibly new) state s_{t+1} is observed.
4. A reinforcement (*reward*) r_{t+1} is received that represents the "goodness" of action a_t.
5. Knowledge of action rewards is updated.

These steps are repeated periodically to learn a policy for optimal application behaviour, with the period configured by a *period-between-actions* parameter. In our approach the observed states in steps 1 and 3 are discrete states or contexts presented to the reinforcement-learning subprocess by the discretisation subprocess. Both of these structures are interpreted by the reinforcement-learning subprocess as *states*. Steps 2, 4 and 5 require individual discussion of action selection, reward, and knowledge update. In addition we discuss the issue of identifying when learning is finished. Our

discussion of reinforcement-learning concepts is at a complexity level necessary to design and use KAFCA. For a more in-depth discussion see (Sutton & Barto, 1998).

3.4.1 Action selection

In the second reinforcement-learning step an action is selected and executed. In applications of reinforcement learning it is common to define the set of states and possible actions in those states at design time based on expert knowledge. This is not possible in our approach as the set of states is unknown at design time. Instead the developer defines a set of available actions and any of these actions can be selected in any state.

Discussions in reinforcement learning distinguish between two conflicting goals of action selection– exploration and exploitation (Kaelbling et al., 1996). The *exploration* goal is concerned with learning the effect of multiple actions in each state. It is achieved by experimenting with actions and observing the associated rewards. Application performance is negatively affected by exploration as some actions that are taken may not be as effective as others. In contrast the *exploitation* goal is to maximise application performance by choosing what are perceived to be the best actions. A consequence of taking the "best" action is that other actions are not executed and knowledge about their usefulness is not learned. The real optimal action may be ignored resulting in suboptimal application performance.

The challenge of balancing these goals so as to optimise long-term application performance is a common dilemma in reinforcement learning, and a variety of strategies have been proposed to address it. These have been surveyed and compared in the literature (Kaelbling et al., 1996; Sutton & Barto, 1998). One obvious strategy is to *greedily* select the "best" action at all times although this obviously means that new knowledge is accumulated very slowly or not at all. The *ϵ-greedy* strategy addresses this by behaving in a near-greedy fashion. Each time an action is selected there is a small probability ϵ that the action is selected randomly. The degree of exploration is controlled by the value for ϵ defined by the developer, and the value reduces over time so that action selection becomes increasingly greedy as knowledge is accumulated. A draw-back of the ϵ-greedy approach is that when selecting an action randomly it is as likely to choose the worst action as the second-best action, i.e., it doesn't exploit its existing knowledge of actions. *Softmax* action selection addresses this by grading the probability of choosing actions. Actions are graded by some function of their current estimated value so that "better" actions have a higher probability of being chosen. The function is parameterised with a value that controls the degree to which better actions are more likely to be chosen. Initially a high value is used and actions are selected almost completely randomly as a result. Over time the value is decreased so that actions are selected more greedily and the application takes advantage of learned knowledge.

Any of these action-selection strategies may be applied to reinforcement-learning problems, and the outcome largely depends on the parameters that are selected (Bowling & Veloso, 2002).

3.4.1.1 Action selection in KAFCA

In our own experiments we observed that the action-selection strategy could affect our perception of when sufficient learning had occurred. The focus of the reinforcement-learning subprocess in KAFCA is on learning an accurate policy so that sensor data can be interpreted. There is no requirement for exploiting learned knowledge during learning. This is not the same focus as action-selection strategies such as ϵ-greedy and softmax, which balance learning against application performance by mixing exploration and exploitation.

In experiments where we applied the softmax strategy we observed that the exploitation element of the strategy caused the perceived-optimal action to be executed more frequently. This reflected the strategy's goal of optimising application performance. However a consequence of this was that other, potentially more optimal, actions were executed less frequently. Once an action was perceived to be optimal the policy was very slow to change to the truly optimal action. It often appeared that sufficient learning had occurred as the policy was changing so slowly.

To overcome this issue we define a simple action-selection strategy that is motivated by the focus on learning an accurate policy rather than exploiting learned knowledge. In KAFCA actions are selected completely randomly to maximise learning. By selecting actions randomly it removes any bias towards particular actions so that the truly optimal policy is learned as quickly as possible. We discuss how the reinforcement-learning subprocess identifies that an accurate policy has been learned later in Section 3.4.3.

3.4.2 Reward

In reinforcement learning "the purpose or goal of the agent is formalized in terms of a special reward signal passing from the environment to the agent" (Sutton & Barto, 1998). The *reward* is a simple number $r_t \in R$ that captures the value of taking the action selected and executed in Step 2. Discussions often refer to it passing from the environment to the application because in reinforcement learning all sensors, including those that sense the reward, are considered to be outside the application. In practice the application senses changes to the environment after taking an action and calculates an appropriate reward internally using a *reward model.* The reward model is a developer-defined, application-specific algorithm for calculating rewards given inputs about the environment. It is the main knowledge-intensive element of reinforcement learning as it encapsulates expert knowledge of the application's

goal and its environment. Alg. 5 shows a simple example of a reward model with conditions and associated rewards. It takes sensor data from a developer-defined set of sensors as input. This input is interpreted as defined by the developer and an appropriate reward is returned.

Algorithm 5: A simple reward model

Input: Environmental changes
Output: Reward
`/* Developer conditions define when good and bad rewards are given */`
if *post-action environment is ...* **then**
| give a good reward;
end
else if *post-action environment is ...* **then**
| give a bad reward;
end

Although the reward model is a knowledge-intensive structure it does not limit the application's ability to learn as other learning techniques do. Techniques that require labeled training data are particularly limited as they can only generalise about sensor data for which they have training examples (Section 2.2.1). In contrast the reward model can be used to learn about sensor data that was not forseen by the developer. The reward model calculates the reward for an action taken in a state. The underlying sensor data that the state represents is irrelevant to the reward model. This enables reinforcement learning to learn a policy that maps *unforeseen* sensor data to actions.

3.4.3 Knowledge update

The knowledge-update step takes the reward r_{t+1} calculated for action a_t in state s_t by the reward model, and updates its knowledge of actions and rewards. Various representations of this knowledge are used in reinforcement-learning. An important characteristic that categorises them is whether they are model-based or model-free (Kaelbling et al., 1996). *Model-based* approaches require the definition of all possible states and state transitions a priori. This obviously requires expert knowledge and introduces inflexibility at run time. In contrast *model-free* approaches do not require such knowledge and are therefore more suited to our knowledge-autonomous approach.

We adopt the model-free representation used in Q-learning (Watkins & Dayan, 1992) as well as its knowledge-update formula. Q-learning stores knowledge in the form of state-action combinations with associated utility values– Q-values. A *Q-value* reflects the learned value of taking an action in a state. The optimal action for a state is identified by comparing the Q-values for each of its possible actions. The update formula for a Q-value is shown in Alg. 6.

A Q-value $Q_{t+1}(s_t, a_t)$ is updated based on its current value ($Q_t(s_t, a_t)$), the observed reward for

Algorithm 6: Q-value update formula

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha[r_t + \gamma(maxQ(s_{t+1})) - Q(s_t, a_t)]$$

an action (r_t), and the long-term value of the action (the maximum Q-value in the transitioned-to state $maxQ(s_{t+1})$). The degree to which a particular update affects the current value is controlled by the α value, also known as the *learning rate*. Similarly the degree to which an update is affected by the long-term value of an action is controlled by the γ value, also known as the *discount rate*. Both values are in the range 0-1 and are specified at design time.

In deterministic environments where actions cause predictable state transitions the learning rate should be 1, as each Q-value update has reliable, consistent information. The real world where context-aware applications are deployed is not deterministic, and a learning rate close to 0 smooths the learning curve by reducing the effect of individual updates. There is an obvious trade-off between the speed (learning rate = ~ 1) and accuracy (learning rate = ~ 0) of learning that is controlled by the developer.

The discount rate is more application specific, as its value depends on whether rewards occur frequently or infrequently. For example, an application that learns a path through a maze can only be rewarded when it succeeds (exits the maze) (Sutton & Barto, 1998). Each action taken on the path contributes to success, but this is only realised once the goal is achieved. For applications in this category the state in which rewards are received is often referred to as the *goal state*. Rewards received in the goal state are propagated by state transitions caused by actions, and the $maxQ(s_{t+1})$ element of the Q-learning update function. A discount rate of ~ 1 places a high value on the long-term benefit of an action, and ensures that rewards are propagated to states that are not goal states. In an application where every action has an immediate reward the long-term value may be less relevant (discount rate = ~ 0).

3.4.3.1 Knowledge update in KAFCA

In our experiments we observed that the learning rate has a significant impact on the accuracy of learned policies when applied to sensor data. One obvious reason for this is that the environment is non-deterministic and therefore a low learning rate should be used, as described above. However we observe another phenomenon that is more specific to learning about discretised sensor data. Discrete states that contain a context edge exhibit a *spatial dependency* between their subspace and their Q-values. Their Q-values depend on where in their subspace rewards are gathered. We categorise

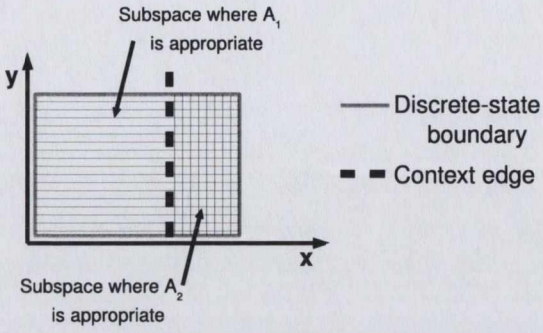

Figure 3.8: An inconsistent discrete state

such states as *inconsistent discrete states* as the meaning of the sensor data they discretise is not consistent– more than one action is optimal within their subspace.

Fig. 3.8 illustrates an example inconsistent state. When sensor values indicate that the environmental situation is on the left side of the context edge the rewards indicate that action A_1 is optimal. On the other side of the context edge rewards indicate action A_2 is optimal. The spatial dependency manifests itself as an oscillation in the Q-values of inconsistent discrete states, e.g., first A_1 appears optimal, then A_2 appears optimal, then A_1 again and so on. The action that appears to be optimal for the discrete state depends on the specific point in its subspace where the action is executed, therefore we class it as a spatial dependency. The true optimal action in an inconsistent discrete state is unknown. If the rewards for both actions are equal then we would expect the action associated with the greatest proportion of the subspace to appear optimal. Otherwise it depends on the relative scale of the rewards as well as the relative proportions of the actions' subspaces. The important thing is that the same action is learned consistently so that a learned policy for discrete states is consistent. The solution to this issue is to use a very low learning rate (~ 0.01) to ensure that the Q-values of a discrete state reflect rewards gathered across its entire subspace, and are not solely influenced by rewards gathered recently in a particular part of the subspace. The scenario in which this phenomenon was observed is discussed in more detail in Section 5.3.

In our experiments where rewards were long term rather than immediate we observed that a *temporal dependency* rather than a spatial dependency affected the accuracy of the learned policy. Long-term rewards are propagated between states by state transitions. The rewards stem from goal states and are propagated outwards from these states by state transitions. However the order in which states are visited is unknown. This order dictates how rewards are propagated and therefore affects the action perceived to be optimal in states. Similarly to the spatial dependency this phenomenon is addressed by using a very low learning rate to smooth the propagation of rewards between states.

The scenario in which this phenomenon was observed is discussed in more detail in Section 5.4.

3.4.4 Stopping learning

Reinforcement-learning-based approaches often do not address the question of when to stop learning because they treat learning as a continuous process. The ratio of exploration to exploitation may decrease over time based on the action-selection strategy (Section 3.4.1) however learning never stops completely. The KAFCA process requires that multiple policies be learned therefore learning must have a definitive end.

It has been shown that Q-learning "converges to optimum action-values with probability 1 so long as actions are repeatedly sampled in all states" (Watkins & Dayan, 1992), but this is under the assumption that an infinite number of actions is taken. The reinforcement-learning subprocess cannot know that a learned policy is truly optimal without taking infinite actions, however it monitors the *stability* of the learned policy to identify when sufficient learning has occurred. At predetermined intervals a new policy is constructed from the current Q-values. This policy is compared to policies constructed at previous stability checks. When a sequence of consistent policies is observed the reinforcement-learning subprocess assumes that an optimal policy has been learned, stops learning and returns the stable policy. The *interval between stability checks* and the *required sequence length* of stable policies are configured by the developer at design time.

This approach to stopping learning based on the stability of a policy is effective as long as the period over which stability is evaluated ($intervalBetweenStabilityChecks * requiredSequenceLength$) is sufficiently long such that any changes in the policy are observed. During our evaluation we observed that where the rate of change of Q-values was too slow a policy could appear to be stable when it was not. This can only be addressed by selecting larger values for *interval between stability checks* and the *required sequence length*. This issue is discussed further in Section 5.5.

3.5 Accurate context definition

Accurate context definition is the first phase of the KAFCA process. It takes sensors as input and outputs context definitions for those sensors. The accuracy of context definitions depends on the ability of discrete states to capture context changes in the environment, therefore this subprocess refines the discrete states that represent sensor data. This occurs at run time to ensure that learned context definitions are accurate for the run-time environment. During the accurate-context-definition phase of KAFCA sensors are processed individually over three steps: initial discrete states are defined

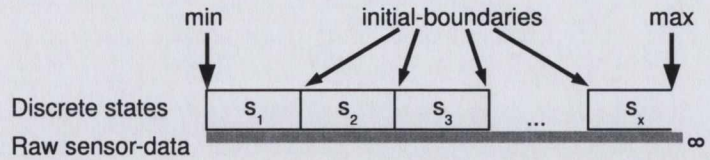

Figure 3.9: Initial discrete states

for a sensor, these discrete states are refined iteratively, and finally context definitions are defined. We now look at these steps in more detail.

3.5.1 Discrete-state initialisation

The discrete-state-initialisation step defines an initial set of discrete states for a sensor prior to learning occurring. Discrete states are necessary to reduce the granularity at which sensor data is provided to the learning algorithm (Section 2.2.2). Sensors produce different sets of values so a generic set of discrete states cannot be used. This step takes a sensor's meta-data and produces a set of discrete states to suit that sensor's data. The meta-data describes a sensor's range of possible values and initial discrete-state boundaries (Section 1.6).

The discrete states for an individual sensor represent one-dimensional ranges of sensor values. During discrete-state initialisation discrete states are created to represent all possible sensor values between the meta-data *min* and *max* values, with boundaries defined by the *initial-boundaries* meta-data (Fig. 3.9). In the event that the *min* or *max* meta-data values of a sensor are $-\infty$ or ∞ the boundary on the first or last state is open-ended. This is the case in Fig. 3.9 where the discrete state s_x has no upper bound. All sensor values greater than the lower bound of s_x's range are mapped to the state. The algorithm for initialising a sensor's discrete states is shown in Alg. 7. The lower bound of the first discrete state is the meta-data *min* value. Each value in the meta-data *initialBoundaries* array represents the upper bound of one discrete state, and also the lower bound of another. The upper bound of the final state is the meta-data *max* value.

3.5.2 Discrete-state refinement

In Section 1.3.1 we identify that fixed discretisation of sensor data may limit the accuracy of learned context definitions. Accurate context definitions depend on discrete states that represent context edges well. Fig. 3.10 illustrates the inaccuracy that is introduced when an inappropriate set of discrete states are used. The illustrated sensor space contains two context edges and is discretised to five discrete states. The first context edge lies within the discrete state s_2 and the second lies on the

Algorithm 7: Discrete-state initialisation algorithm

Input: Sensor meta data
Output: Initial discrete states
`/* Set the lower bound for the first discrete state */`
lowerBound = meta_data.min;
foreach *b in meta_data.initialboundaries* **do**
  upperBound = *b*;
  create discrete state with range lowerBound to upperBound;
  lowerBound = upperBound;
end
`/* create the final discrete state */`
create discrete state with range lowerBound to meta_data.max;

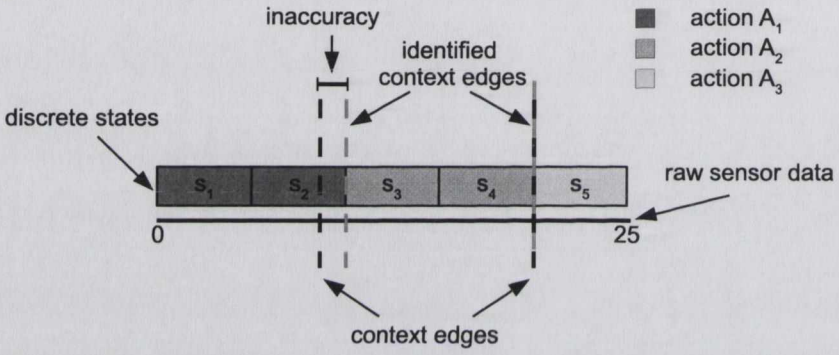

Figure 3.10: Inappropriate discrete states cause inaccurate context definitions

boundary between s_4 and s_5. Context definitions are expressed at the granularity of discrete states, therefore the first context edge is identified at the boundary between the dissimilar states s_2 and s_3. This introduces some inaccuracy into a context definition due to the difference between the identified and actual context-edge location.

In this example s_2 is an *inconsistent discrete state*, as described in Section 3.4.3, as it discretises sensor data that does not have consistent meaning. Inconsistent discrete states introduce inaccuracy in context definitions as they hide the true context-edge location. In contrast the second context edge lies at the boundary between states s_4 and s_5. The perceived context edge correlates with the real context edge and therefore the discrete states introduce no inaccuracy. States s_4 and s_5 are both *consistent states* as the context edge lies at their boundaries and not within their subspaces.

The discrete-state-refinement step takes the initial discrete states of a sensor and iteratively refines them so they better capture context edges. In order to achieve this KAFCA must identify inconsistent discrete states and split their subspaces. The current version of KAFCA refines the one-dimensional discrete states of individual sensors in the accurate-context-definition phase. However, this refinement

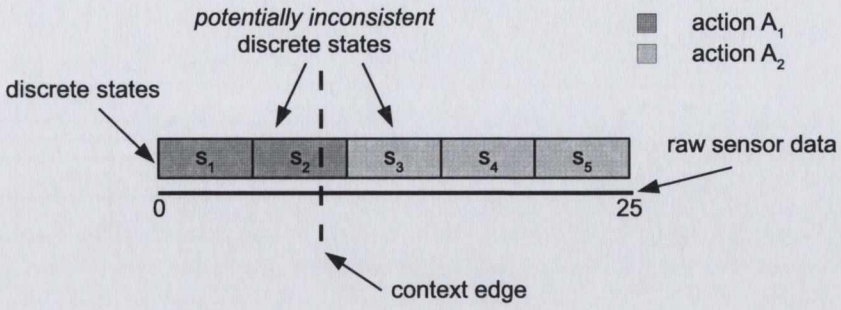

Figure 3.11: Potentially-inconsistent states

approach could also be applied to n-dimensional discrete states.

3.5.2.1 Inconsistent-discrete-state identification

Inconsistent discrete states are those discrete states that contain context edges. The true location of context edges (and therefore the true inconsistency of states) cannot be known without knowledge of how every sensor value influences action selection. As discussed earlier it is infeasible to learn about sensor data at this granularity therefore KAFCA can only reason about the *potential inconsistency* of discrete states. Potentially-inconsistent discrete states are those discrete states that have dissimilar neighbours (Fig. 3.11). The reinforcement-learning subprocess learns a policy for the set of discrete states currently being refined. This policy is used to compare neighbouring discrete states and identify dissimilar neighbours.

In the example in Fig. 3.11 the true context-edge location may be in the subspace of either discrete state s_2 or s_3, so both are potentially inconsistent. The subspaces of both s_2 and s_3 are therefore split.

3.5.2.2 Discrete-state-subspace splitting

Each potentially-inconsistent discrete state represents sensor data in n dimensions however it may have a context edge in only one/some of those dimensions. Fig. 3.12(a) shows a two-dimensional sensor space that has a context edge in each dimension. The sensor space is represented by twenty discrete states (s_1-s_{20}). Some states such as s_1, s_6 and s_{11} have no dissimilar neighbours therefore they are not potentially inconsistent. A number of discrete states have dissimilar neighbours in just one dimension. States s_2, s_3, s_7, s_8 and s_{12} have dissimilar neighbours along the x-axis, while states s_{14}, s_{15}, s_{18}, s_{19} and s_{20} have dissimilar neighbours along the y-axis. These discrete states only need to be split along the axis in which they have dissimilar neighbours to improve the accuracy with which

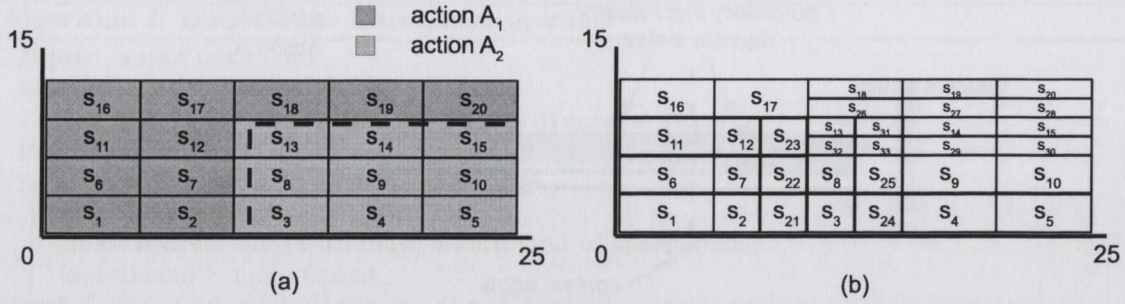

Figure 3.12: Discrete-state-subspace splitting

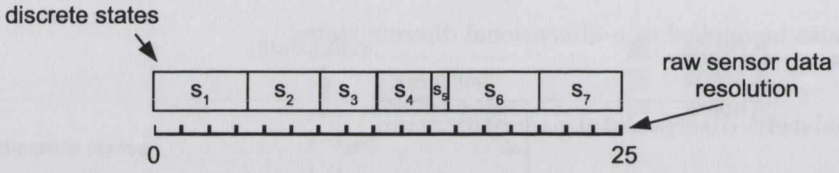

Figure 3.13: Discrete states that are too small represent no sensor values

they capture context edges. State s_{13} has dissimilar neighbours along both axes and therefore must be split in both dimensions. Fig. 3.12(b) shows the discrete states that are output after inconsistent states have been split. An inconsistent state with dissimilar neighbours in m of its n dimensions is split into 2^m discrete states after refinement.

Inconsistent discrete states are split as long as the split subspaces represent sensor data. Subspaces smaller than the granularity of the sensor data do not represent any values within the sensor space. Fig. 3.13 shows a situation where a discrete state (s_5) does not represent any sensor values, as the sensor space it represents lies *between* possible sensor values. This situation is unlikely to occur for sensors that have a high degree of precision however it could occur for sensors with a coarse granularity, e.g., the granularity of a road-traffic sensor would be a single car. To guard against this case each sensor is annotated with a meta-data value for sensor *precision* (Section 1.6) and this value is considered during discrete-state splitting.

Fig. 3.14 shows the progression of a set of discrete states as they are refined over a number of iterations. At each iteration the subspaces of potentially-inconsistent discrete states are split so they capture context edges more accurately. The number of refinement iterations is configured by the developer at design time.

The algorithm for refining discrete states is shown in Alg. 8. For each refinement a new policy is learned for the current set of discrete states. The inconsistency of each discrete state is then evaluated

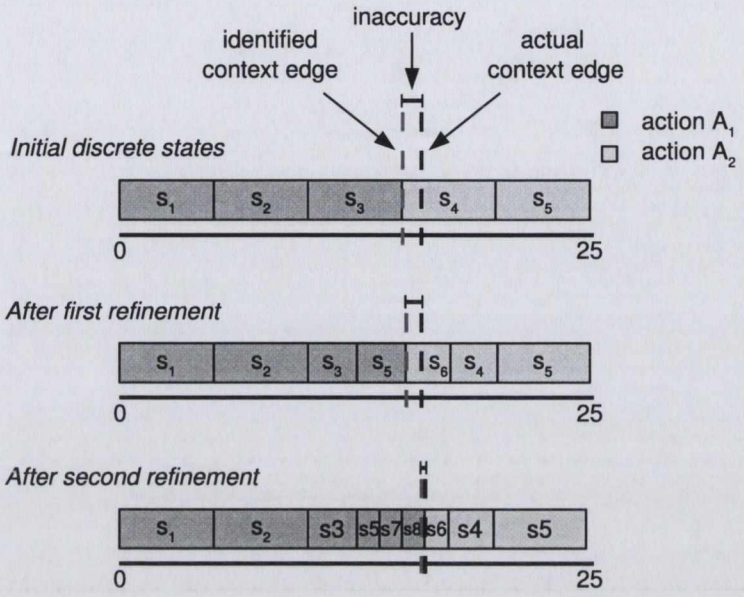

Figure 3.14: Discrete-state refinement

by comparing its optimal actions to those of its neighbours. If it is inconsistent then it is compared to its neighbours in each dimension. If the neighbours in that dimension are different, and its range is sufficiently large, then its subspace is split in that dimension.

We describe the operation of this algorithm using the temperature sensor from the example discussed at Alg. 1. The algorithm refines an initial set of discrete states such as 0-32, 32-64, 64-100°C for the sensor. In the first iteration it learns a policy for these discrete states, and the policy indicates that the shutters should close for the first discrete state (0-32°C), while all other discrete states cause them to open. This indicates that there is a context edge in one of the first two states, and these states are therefore split into four states– 0-16, 16-32, 32-48, 48-64°C. This brings the boundary between discrete states in the discretisation layer closer to the context edge (11°C) In the second iteration a policy is learned that indicates that 0-16°C now causes the shutters to close, while all others cause them to remain open. As a result the discrete states 0-16°C and 16-32°C are split into four states to further refine knowledge of the context-edge location. This process continues for a predetermined number of iterations.

3.5.3 Context definition

Contexts are defined when the number of refinements specified by the developer have been carried out. As described in Section 3.3 contexts are collections of discrete states and their subspace is defined

Algorithm 8: Refine inconsistent discrete states

```
Input: Discrete states
Output: Refined discrete states
foreach refinement r do
    learn policy for discrete states using reinforcement learning;
    foreach discrete state d do
        get optimal action for d from policy;
        select neighbouring discrete states of d;
        get optimal actions for neighbours;
        /* check if the discrete state is inconsistent */
        if action for d != neighbour actions then
            /* check in which dimensions it should be split */
            foreach dimension dim of d do
                select neighbouring discrete states in dim;
                if action for d != dim neighbour actions
                and range in dim >= 2*meta_data.precision then
                    split d in dimension dim;
                end
            end
        end
    end
end
```

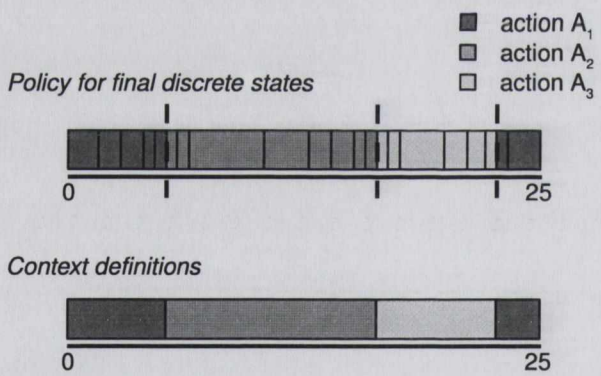

Figure 3.15: Similar discrete states are combined in contexts

by the combined subspaces of their discrete states. The context-definition step compares discrete states to their neighbours based on a learned policy. Dissimilar neighbours indicate context edges and similar neighbours indicate context spaces. Fig. 3.15 shows how similar neighbouring discrete states in one-dimensional space are combined in contexts.

The current version of KAFCA carries out context definition on individual sensors during the accurate-context-definition phase, however the approach also facilitates context definition for n-dimensional discrete states, i.e., combined sensors.

The algorithm for combining discrete states in contexts is shown in Alg. 9. A policy is learned for the finalised set of discrete states after refinement. Each discrete state is compared to its neighbours to evaluate which context it should be assigned to. If the discrete state is similar to a neighbour and that neighbour is in a context then it is added to the context. If the discrete state has multiple similar neighbours, in multiple contexts then those contexts are joined together to form one contiguous context. If none of its similar neighbours are in contexts, or it has no similar neighbours, then a new context is created and the discrete state is added.

Again we describe this algorithm using the temperature sensor from earlier discussions. For this sensor, due to a context edge at 11°C, the significant discrete states after refinement are located near this point in the sensor space. After 4 refinements by Alg. 8 these states would be ..., 6-8, 8-10, 10-12, 12-14, ... A learned policy for these states would show that states that represent values below 10°C cause the shutters to close while those above 12°C cause the shutters to open. For the purposes of this example we assume that the discrete state 10-12°C also causes the shutters to close, although this is uncertain due to the context-edge location at 11°C. Alg. 9 compares discrete states to identify context-edge locations and define contexts. Starting at the first discrete state (0-4) it compares it to its neighbours. For each neighbour, in this case on the discrete state 4-6, it compares their actions in the policy. 4-6 has the same action but does not belong in an existing context definition therefore 0-4 is added to a new context definition. The discrete state 4-6 is then compared to each of its neighbours, finds that the similar neighbour 0-4 is already in a context definition and is assigned to that context definition. This continues up to and including 10-12, as these similar, neighbouring discrete states are all added to the same context definition. Beyond this point the discrete state 12-14 finds that it has no similar neighbours in existing context definitions and is added to a new context definition. As the algorithm progresses all remaining discrete states are added to this definition. The outcome is a pair of context definitions that represent the sensor space 0-12°C and 12-100°C respectively.

3.6 Sensor selection

Sensor unavailability or unsuitability may limit the ability of a context-aware application to identify its context. The sensor-selection subprocess is responsible for selecting the most suitable sensors for identifying the context from those available to the application at run time. The sensor-selection phase of KAFCA commences when the accurate-context-definition subprocess has completed. It takes the context definitions of individual sensors as input and outputs the most suitable set of sensors for an application.

Algorithm 9: Context-definition algorithm

```
Input: Discrete states
Output: Contexts
learn policy for discrete states using reinforcement learning;
foreach discrete state s do
    get optimal action for s from policy;
    select neighbouring discrete states of s;
    foreach neighbour state n do
        get optimal action for n from policy;
        /* check if they should be in the same context */
        if action for s == action for n then
            /* if the neighbour is in an existing context */
            if n belongs to existing context then
                if s is not in an existing context then
                    add s to context of n;
                end
                else
                    join context of s and context of n;
                end
            end
        end
    end
    /* if not in a context then create a new context */
    if s is not in an existing context then
        create new context with s as first member;
    end
end
```

The sensor-selection subprocess applies a similar approach to those used by (Albinali et al., 2007) and (Krause et al., 2006) to select the structure of Bayesian networks (discussed in Chapter 2). The structure of a Bayesian network defines how variables (nodes) are related to each other. The approaches of (Albinali et al., 2007) and (Krause et al., 2006) select the structure of Bayesian networks so that they best identify high-level contexts. They identify the best structure by iteratively extending the network (adding new nodes) and then evaluating the new structure. The probabilities for each version of the network structure are learned using training data, and each network's accuracy at identifying contexts is measured. The accuracy is calculated by comparing the outputs of the trained network to the correct outputs in the training data. They select the network structure that provides highest accuracy.

The sensor-selection subprocess similarly identifies the best sensors by iteratively combining them and evaluating their accuracy for identifying contexts. In our case accuracy is measured in terms of application performance, based on the intuition that more suitable sensors lead to better decision making and better application performance. The approaches of (Albinali et al., 2007) and (Krause

et al., 2006) can only evaluate the accuracy of networks that use sensors for which they have labeled training examples. In contrast our evaluation is more generic. It is independent of the underlying sensor data used to identify contexts, therefore it can be used to evaluate sensors that were unforeseen by the developer at design time.

In this subprocess sensors are combined and evaluated over three steps: a new combination of sensors is selected, their context definitions are combined, and the combination is evaluated.

In our example application there are five available sensors. By this phase of KAFCA each of these sensors has its own context definitions based on its refined discrete states. The challenge now is to identify the most suitable combination of these sensors for identifying application contexts. The first step is to identify the best individual sensor for the application, which will serve as a base for other combinations of sensors. A policy is learned for each individual sensor and application performance is evaluated while reacting based on this policy. For example, the application using solely the temperature sensor would cause the blinds to close when temperatures were below 12°C and open otherwise. The performance of the application, given this pattern of behaviour, is measured using a developer-defined metric. By comparing the performance of each sensor the best individual sensor is identified and this sensor becomes the base sensor, in this example we assume the temperature sensor provides the best context information and therefore causes the best application behaviour.

In a second iteration each of the other four sensors is then combined individually with the temperature sensor. Each pair's possible values form a 2-dimensional sensor space, and their individual context definitions are combined to form 2-dimensional rectangular context definitions. A policy is again learned for each combination and application performance while using the policy is evaluated against the metric. If application performance while using a sensor combination surpasses the performance of the individual temperature sensor then that combination becomes the new base. In this example the combination of temperature sensor and light-intensity sensor may surpass the individual temperature sensor.

In a third iteration the three remaining sensors are each combined individually with the two base sensors. Each new combination forms a 3-dimensional sensor space and 3-dimensional context definitions. Again a policy is learned for each three-sensor combination and application performance is evaluated with each policy. In this example the performance of none of these combinations surpass the two-sensor combination of temperature and light-intensity sensor, therefore the sensor-selection process has completed and the most suitable sensors have been found.

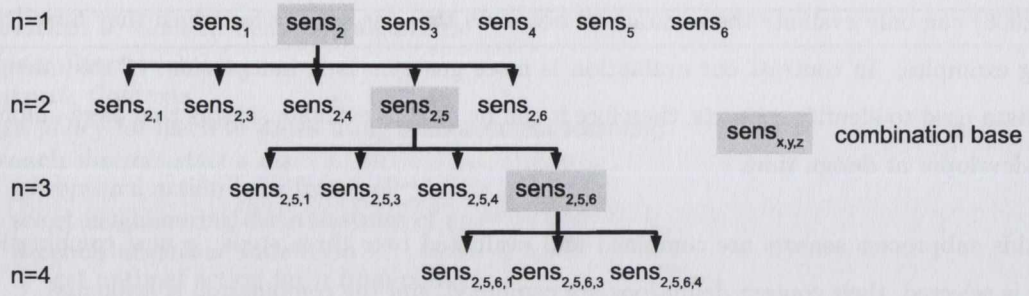

Figure 3.16: Search for best sensor combination

3.6.1 Combination selection

The combination-selection step selects the next set of sensors to be combined and evaluated. It performs an informed search to identify the most suitable combination of sensors for the application. The search is informed by measurements of application performance in the combination-evaluation step, which we will discuss in Section 3.6.3.

The example in Fig. 3.16 illustrates the informed-search algorithm. The search begins with single sensors (n=1). Each sensor is selected individually and its evaluation result is used to identify the most suitable single sensor for identifying application contexts. This sensor becomes the base for combinations of size n=2. These combinations are selected by taking the base sensor and combining it with each other available sensor. The best-performing pair becomes the new base for future combinations of size n=3, the best performing trio becomes the new base for future combinations of size n=4, and so on. Combinations are extended until the performance of a base combination is not exceeded by those of its child combinations, in this case when n=4.

The search algorithm is shown in Alg. 10. An initially empty *baseSensors* set of sensors is defined. For each iteration the context definitions of each available sensor is combined with the definitions of the current base sensors to form a set of child context definitions. These definitions are evaluated, and the best-performing set of child sensors is identified. If the best child sensors perform better than the current base sensors then they become the new base.

During sensor selection all sensors are treated equally. This is based on the intuition that any sensor may provide useful and interesting information to any application, and limiting the sensors considered may cause information loss. However in the interest of increasing the efficiency of the search it might be useful to prevent some sensors being considered. SensorML (Open Geospatial Consortium, 2000) is a standard for sensor meta data. In KAFCA it might be used to discard sensors from consideration in two ways. Where multiple sensors of the same type are available it could be used

to discard those sensors that provide less accurate sensor data. It might also be used to discard sensors of a similar type to those that have already been evaluated and found unsuitable by an application. For the purposes of this thesis we do not employ such an approach as our focus is on maximising the information available to applications so they can accurately identify their context.

Algorithm 10: The combination-search algorithm

```
Input: Evaluation results
Output: sensors
baseSensors is an empty set of sensors;
basePerformance = -1;
while new baseSensors found and not all sensors combined do
    foreach sensor s do
        bestChildSensors is an empty set of sensors;
        bestChildPerformance = -1;
        if s not in baseSensors then
            /* combine using the context-combination step */
            childContexts = context combination(s, baseSensors);
            /* evaluate using the combination-evaluation step */
            childPerformance = combination evaluation(childContexts);
            /* check if this is the best performing child */
            if childPerformance > bestChildPerformance then
                bestChildPerformance = childPerformance;
                bestChildSensors = s + baseSensors;
            end
        end
    end
    /* check if the best child is better than the current base */
    if bestChildPerformance > basePerformance then
        basePerformance = bestChildPerformance;
        baseSensors = bestChildSensors;
    end
end
/* The final set of base sensors are most suitable */
return baseSensors;
```

3.6.2 Context combination

The context-combination step takes the one-dimensional contexts of individual sensors and combines them to form n-dimensional contexts. The contexts of sensors are combined by extending each context of one sensor with each context of the others. Fig. 3.17 illustrates how the individual contexts of two sensors ($sens_1$ and $sens_2$) are combined to create a set of two-dimensional contexts. These contexts represent the combined sensor space of both sensors. Three dimensional contexts contexts are created by combining a set of two-dimensional contexts with the contexts of a third sensor. The algorithm

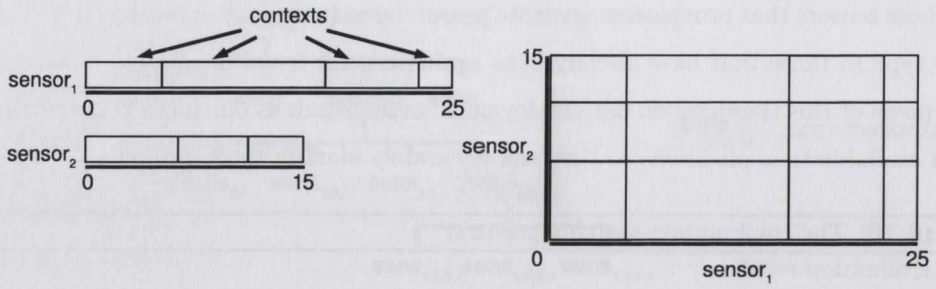

Figure 3.17: Combining contexts

for combining contexts is shown in Alg. 11.

Algorithm 11: Context-combination algorithm

```
Input: sensor combination
Output: n-dimensional contexts
/* initialise the combined contexts */
combinedContexts = the contexts of the first sensor;
/* iteratively extend these contexts with the contexts of the other sensors */
foreach other sensor s in combination do
    temp is an empty set of contexts;
    foreach context c of combinedContexts do
        /* extend the combinedContext with each context of s */
        foreach context cc of s do
            extend context c with context cc;
            add extendedContext to temp;
        end
    end
    combinedContexts = temp;
end
```

3.6.3 Combination evaluation

The combination-evaluation step evaluates the suitability of sensors for identifying the context, which informs the search for the best sensor combination. Sensor suitability is relative to the application and its goals, therefore we cannot define a generic measure of suitability. Instead it is measured using an application-specific metric for application performance. The better the application performs while using a set of sensors to identify its contexts the more suitable those sensors are for identifying application contexts. Similar to the reward model (Section 3.4.2) the metric uses sensor data to observe and record application performance. It then calculates a simple number $r \in R$ that captures the overall performance of the application.

The combination-evaluation step observes application performance over an evaluation period. In order to define how the application should behave a policy is learned for the n-dimensional contexts of the current sensor combination using reinforcement learning. This policy maps contexts to application actions. During the evaluation period the application periodically discretises sensor data to an n-dimensional context, performs a lookup on the appropriate action for that context in the policy, and executes the action. The application-specific metric for application performance is also updated and may gather data specified by the developer. At the end of the evaluation period the metric produces a numeric representation of the application's performance while using the particular combination of sensors, and this is the value that informs the search for the best combination. The algorithm for combination evaluation is shown in Alg. 12.

Algorithm 12: Combination-evaluation algorithm

```
Input: N-dimensional context definitions
Output: Evaluation result
learn policy for contexts using reinforcement learning;
for i=0 i<evaluation period i+=period between actions do
    record developer-defined data for metric;
    discretise sensor data to context c;
    get optimal action a for c from policy;
    execute a;
end
calculate metric result;
```

3.6.4 Critique

This approach to identifying the most suitable set of sensors is effectively a hill-climbing search. A hill-climbing search is a heurestic search that always aims to improve on the existing solution with the next solution– it never selects a less effective solution Callan (2003). As a result it may find a local minima and fail to find the true optimal solution. This could potentially occur in our algorithm if the most suitable sensors were only effective when combined together and not effective when combined with other sensors, e.g., in Fig. 3.16 the combination $sens_{1,3}$ is not evaluated. However both sensors are evaluated individually in $sens_1$ and $sens_3$, and on multiple other occasions with other sensors, e.g., $sens_{2,1}$ and $sens_{2,3}$, therefore we consider this search sufficient for identifying the most suitable set of sensors in most cases.

3.7 Chapter summary

This chapter described the design of the approach to knowledge-autonomous context awareness proposed in this thesis. Initially we discussed existing definitions of context and their relevance and limitations for knowledge-autonomous applications. A novel definition of context from the application's perspective was then described, which is independent of human symbolism. This lead into a description of how sensor data and contexts should be represented so that an application could adjust its context definitions. We described a flexible representation of sensor data based on a sensor space, and a flexible representation of context based on context edges.

The KAFCA process that realises this theory of knowledge-autonomous context was then described. At the core of the process is reinforcement learning, which is used to learn policies that map discretised sensor data to application actions. The issues of action selection and knowledge update are common challenges in reinforcement learning, and these issues were discussed from the perspective of learning about sensor data. A flexible discretisation process for sensor data was also discussed.

The KAFCA process uses reinforcement learning to learn accurate context definitions and select the most suitable sensors for an application. The accurate-context-definition subprocess addresses the challenge of defining contexts that are accurate for the run-time environment, by refining how sensor data is discretised. An initial set of discrete states to represent the sensor data of a sensor are defined using sensor meta data. These discrete states are iteratively refined by learning a policy that describes how they influence action selection, identifying inconsistent discrete states that may contain context edges, and splitting these discrete states. Context definitions for each sensor are defined when refinement finishes.

The sensor-selection subprocess addresses the challenge of selecting suitable, available sensors at run time. The process searches for the best sensor combination using an informed-search algorithm– new sensor combinations are selected based on the performance of previous combinations. The context definitions of individual sensors are combined to form n-dimensional context definitions, and application performance is measured while using these context definitions to adapt its behaviour. The best set of sensors for inferring the context in a particular run-time environment is iteratively identified using this process.

Chapter 4

Implementation

The previous chapter describes the design of our approach to learning context definitions at run time, and discusses how its techniques and algorithms combine to addresses the issues of context misinterpretation as well as sensor unavailability and unsuitability.

This chapter describes the C++ implementation of these techniques and algorithms using diagrams in Unified Modelling Language (UML) notation and code where necessary. The UML was selected due its recognition as a modelling-language standard and also due to the author's familiarity with it. We use *class* diagrams to capture the information and interfaces of elements within the implementation, and *sequence* diagrams to capture the interactions and collaborations between these elements at run time. C++ was selected as the programming language of the implementation due to the availability of a reinforcement learning framework which was extended by this implementation. This framework is discussed further in Section 4.6.

We begin the chapter with a high-level overview of the implementation that is structured around different groups of functionality for discretisation, reinforcement learning, accurate context definition, and sensor selection, and we describe the sequence of interactions between these functional groups during the KAFCA process. This is followed by descriptions of the classes and their interactions within these functional groups. We also discuss configuration details as well as the implementation of the most important operations.

4.1 Overview

The implementation of KAFCA described in this thesis is divided into a number of functional groups. Fig. 4.1 shows an overview of the architecture. At the top level are *context-aware applications.*

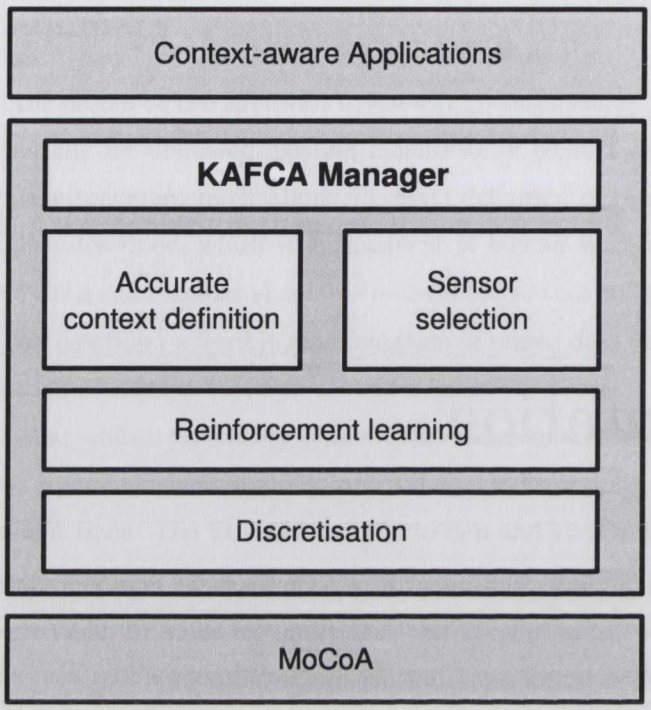

Figure 4.1: High-level architecture

Each application defines configuration parameters and application-specific implementations of required classes, e.g., the possible application actions and the reward model for evaluating actions. The *KAFCA manager* provides an entry point for applications to the KAFCA process and acts as a coordinator of the other functional groups. The *accurate-context-definition* and *sensor-selection* groups implement the algorithms described Sections 3.5 and 3.6 of the design chapter. The *reinforcement-learning* group is responsible for learning policies that define the relationship between sensor data and actions, and the *discretisation* group manages the mapping from sensor-space tuples to discrete states and contexts. At the bottom of the architecture is the *MoCoA* middleware, which handles interactions with sensors and actuators.

Fig. 4.2 illustrates the sequence of interactions that occur between these functional groups during the KAFCA process. As shown in the figure the application interacts only with the KAFCA manager, which coordinates the other functional groups of the implementation. The functionality within each group is accessed through a single class that coordinates interactions with other classes.

A number of inter-related classes are commonly used across most of the functional groups (Fig. 4.3). Both the *DiscreteState* and *Context* classes represent sets of sensor-data tuples in the sensor space. The *DiscreteState* class encapsulates a vector of *Range* instances that represent its dimensions

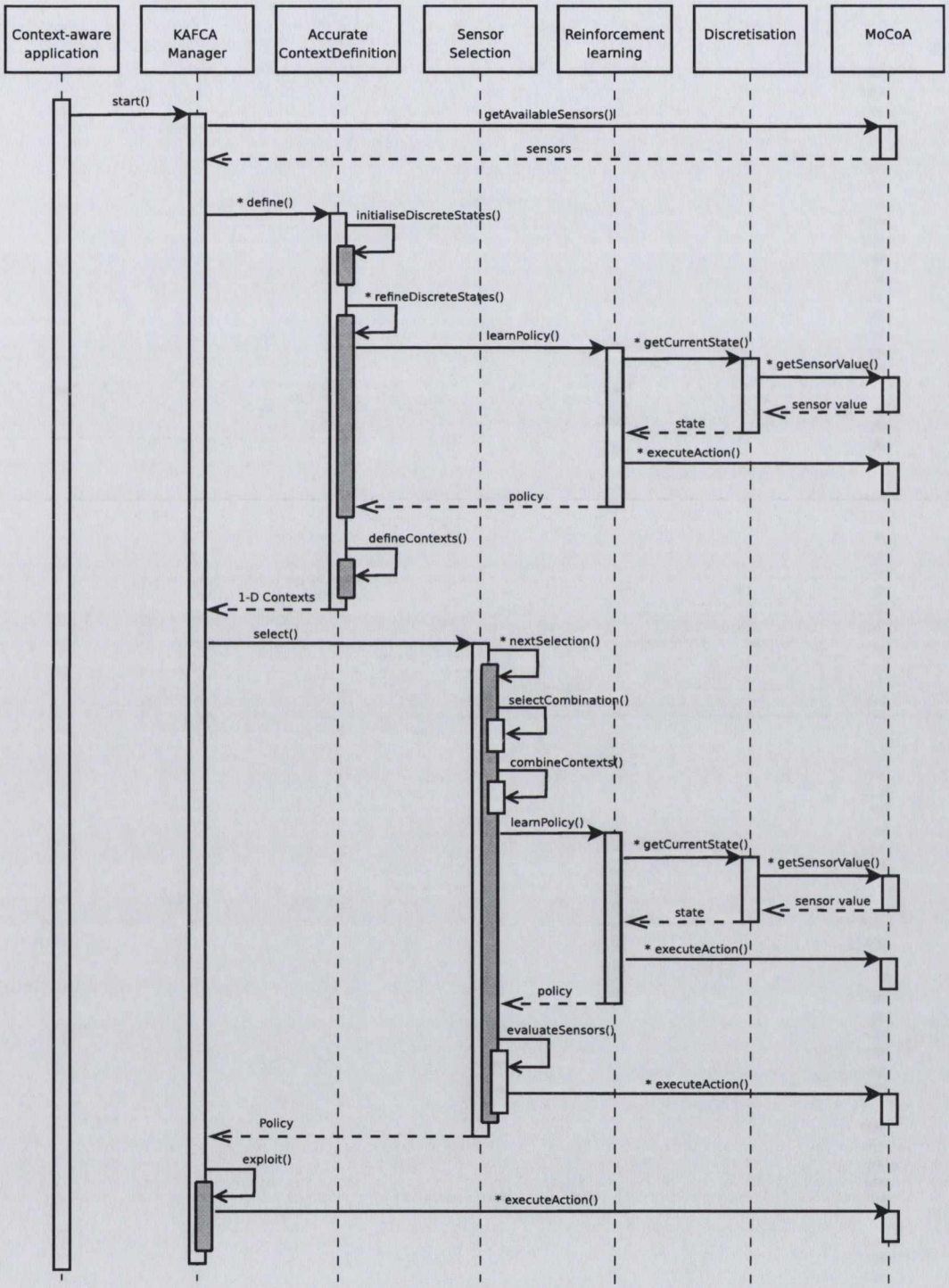

Figure 4.2: High-level sequence of operations

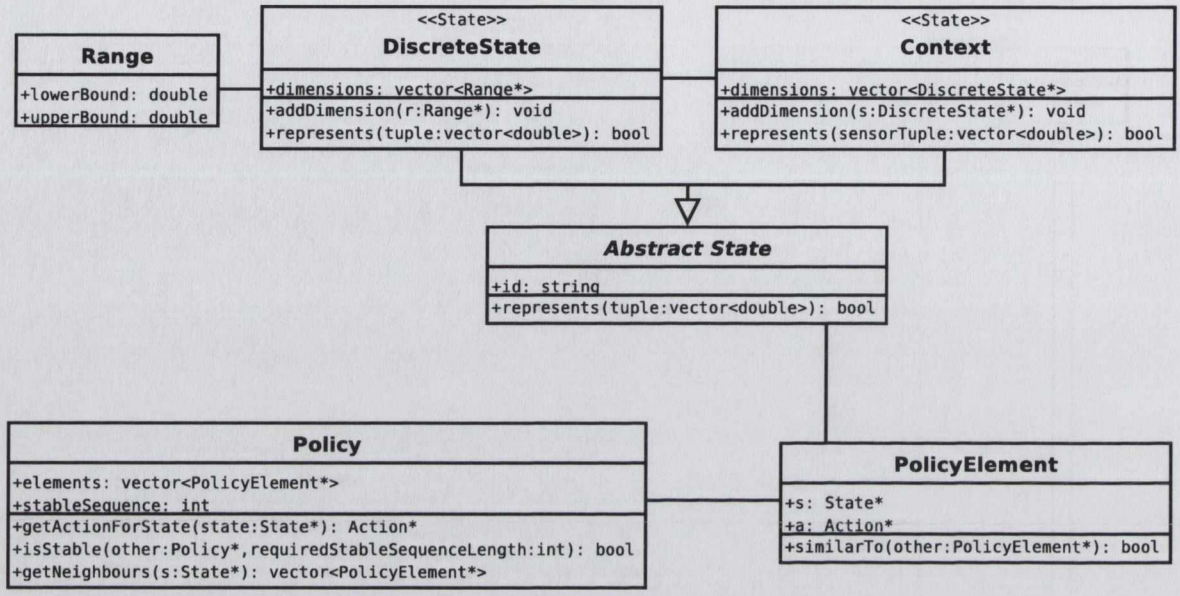

Figure 4.3: Commonly-used classes

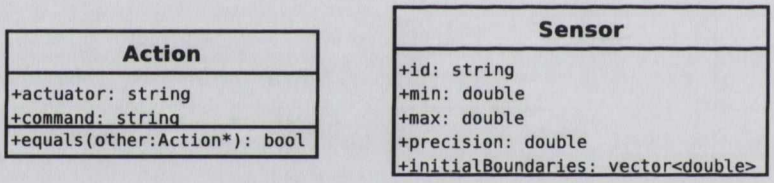

Figure 4.4: The *Sensor* and *Action* classes

in the sensor space. Each *Range* defines a lower and an upper bound for a particular dimension of the sensor space. The *Context* class in turn encapsulates a vector of *DiscreteStates* that represent its dimensions in sensor space. Both classes extend the abstract *State* class, which is used internally by the reinforcement-learning functional group. The *Policy* class encapsulates learned knowledge of mappings between states and actions in *PolicyElements*. We now describe the functional groups within the architecture in more detail.

4.2 Context-aware application

The context-aware application encodes developer knowledge at the design phase of the application lifecycle. It defines application-specific structures and parameters that are required by KAFCA. The *Action* class (Fig. 4.4) encapsulates *actuator* and *command* attributes. The *actuator* attribute stores the name of an actuator and the command attribute defines the change to the actuator that the *Action*

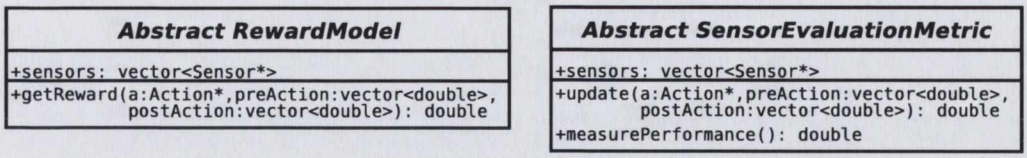

Figure 4.5: The abstract *RewardModel* and *SensorEvaluationMetric* classes

causes. The application must include a set of *Action* instances, one per possible application action, that set the values of these attributes.

The abstract *RewardModel* class (Fig. 4.5) defines a *getReward()* method for calculating the reward during reinforcement learning. This method takes the action, as well as sensor-data tuples for before and after action execution as parameters. The set of sensors that are required for calculating the reward are specified in the *sensors* attribute of *RewardModel*. These sensors are used to select sensor data for the *preAction* and *postAction* sensor-tuple parameters of *getReward()*. The application must include a subclass of *RewardModel* that specifies required sensors for calculating the reward in the *sensors* attribute and implements the *getReward()* method.

The abstract *SensorEvaluationMetric* class (Fig. 4.5) defines the methods *update()* and *measurePerformance()*. The *update()* method is used to periodically update a record of application performance across an evaluation period by interpreting *preAction* and *postAction* sensor data, and the *Action* taken. Similar to the *RewardModel* these sensor-data tuples are populated with sensor data from the sensors specified in the *sensors* attribute of this class. The *calculateResult()* method uses recorded data to calculate some application-specific measure of performance. The application must include a subclass of *SensorEvaluationMetric* that implements these methods and sets the required sensors in the *sensors* attribute.

The sensors specified in the *RewardModel* and *SensorEvaluationMetric* classes are instances of the *Sensor* class (Fig. 4.4). The *id* attribute of the class stores a unique identifier that matches the name of a physical sensor, and the other attributes represent sensor meta-data required by the KAFCA process. In the context-aware application only the *id* attribute of each *Sensor* instance needs to be specified by the developer. The other attributes are populated from sensor meta-data at run-time.

In addition to these classes the application also defines parameters that configure the KAFCA process. Reinforcement-learning requires *learningRate* and *discountRate* parameters to configure the Q-learning update function. The *periodBetweenActions* parameter must also be defined. Our approach to identify when learning is complete based on policy stability requires parameters for *updatesBetweenStabilityTests* and *requiredStableSequenceLength*. The discrete-state-refinement process requires a *numberOfRefinements* parameter that defines the number of refinements to carry out for each sensor's

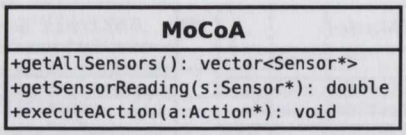

Figure 4.6: The *MoCoA* class

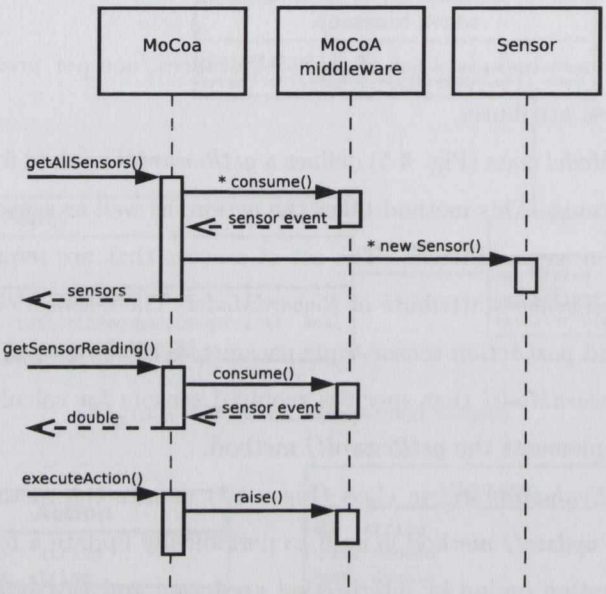

Figure 4.7: *MoCoA*– operations

discrete states. Finally the sensor-selection process requires a *numberOfEvaluationActions* parameter that defines how many actions should be taken when evaluating each sensor combination.

The context-aware application makes a single *start()* method invocation on the KAFCA manager, which takes these structures and configuration values as parameters.

4.3 MoCoA

As introduced in Chapter 1 MoCoA is a middleware that provides a set of programming abstractions and services for building context-aware applications (Senart et al., 2006). MoCoA supports a set of programming abstractions that are suitable for building a wide range of context-aware applications. For each abstraction it provides a library of components that can be integrated into applications. In MoCoA applications are structured around the sentient object abstraction (Biegel & Cahill, 2004), where sentient objects are intelligent components that interpret and use context information from sensors and other sentient objects to drive their behaviour. In this model a *sensor* is perceived to be a

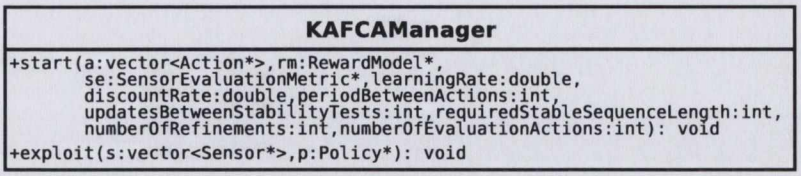

Figure 4.8: *KAFCAManager* class

software-event producer that produces events in response to its observations in the real world. Sentient objects affect the environment through *actuators*, which are devices that consume software events and react by attempting to change the environment. MoCoA uses the STEAM event service (Meier & Cahill, 2003) to support this event-based communication. In this framework KAFCA delegates its sensor and actuator communication and discovery to the MoCoA abstractions.

Fig. 4.6 shows the *MoCoA* class. The *MoCoA* class provides methods for sensor and actuator interaction that are required by other processes, and delegates communication to the MoCoA middleware. The middleware uses the STEAM event service (Meier & Cahill, 2003) to support event-based communication with sensors and actuators. STEAM events consist of a subject and an attribute list of name–value pairs. Sensors are STEAM event producers and actuators are STEAM event consumers. The MoCoA middleware provides a *consume()* method for consuming events from STEAM and a *raise()* method for raising events over STEAM.

Fig. 4.7 illustrates the operations and interactions that occur within the *MoCoA* class. The *MoCoA* class's *getAvailableSensors()* method creates and returns a vector of *Sensor* instances based on sensor-data consumed by the MoCoA middleware. The *getSensorReading()* method retrieves a sensor reading for the parameterised *Sensor* via the MoCoA middleware. The *executeAction()* method raises the event specified by the *Action* parameter.

4.4 KAFCA manager

Fig. 4.8 shows the *KAFCAManager* class. The *KAFCAManager* class provides an entry point for the context-aware application to the KAFCA process. Fig. 4.9 illustrates the sequence of operations and interactions that occur within the *KAFCAManager*. The process commences when the *start()* method is invoked by the context-aware application. The first operation is to retrieve the set of available sensors by invoking *getAvailableSensors()* on the *MoCoA* class, which returns a vector of *Sensor* instances. The *KAFCAManager* then delegates to the *AccurateContextDefinition* class to define *Contexts*. The *define()* method is called once for each *Sensor* and returns one-dimensional *Contexts* for the particular

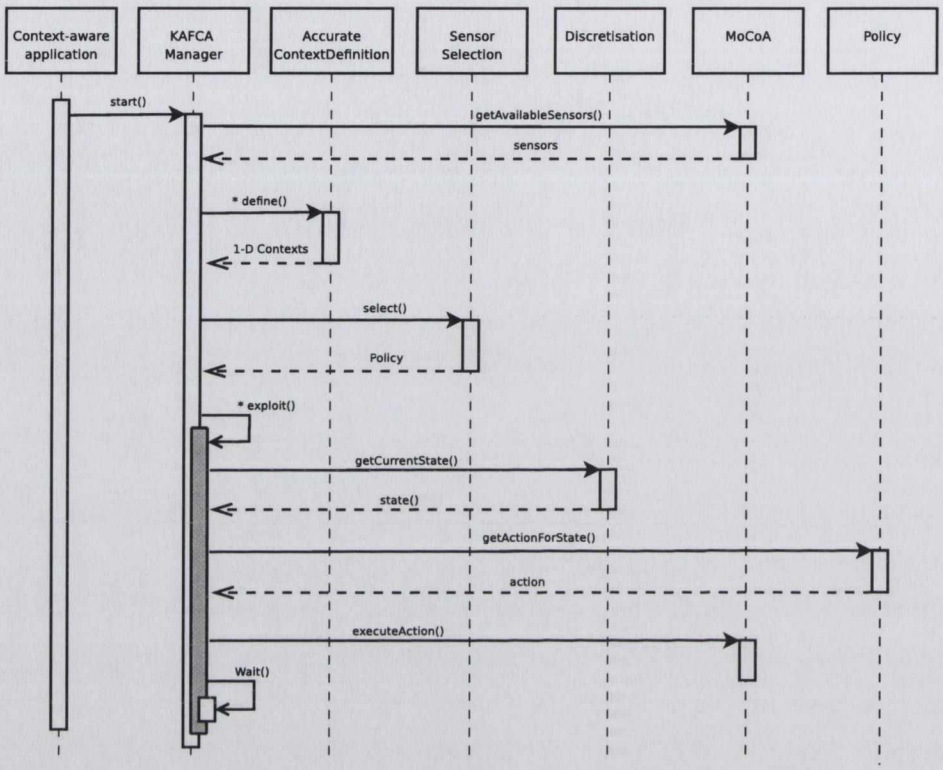

Figure 4.9: *KAFCAManager*– sequence of operations

sensor. When all sensors have been processed individually the *KAFCAManager* delegates to the *SensorSelection* class to select the best set of sensors for the application. The *select()* method returns the *Policy* for those sensors. Finally the *KAFCAManager* class self-invokes its *exploit()* method (Listing 4.1). This method retrieves the current state from the *Discretisation* class, selects the action associated with that state from the *Policy*, and executes the action using the *MoCoA* class. The *exploit()* method iterates every *periodBetweenActions* seconds until the application is undeployed.

```
1 //Get the current state of the environment
2 State st = Discretisation->getCurrentState(sensors);
3 //Select and execute the appropriate action for this state
4 Action a = policy->getActionForState(st);
5 MOCOA->executeAction(a);
6 //Sleep for an application-specific period between actions
7 Sleep(periodBetweenActions);
```

Listing 4.1: The `exploit()` method

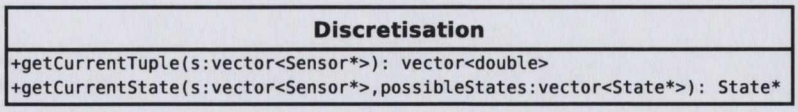

Figure 4.10: The *Discretisation* class

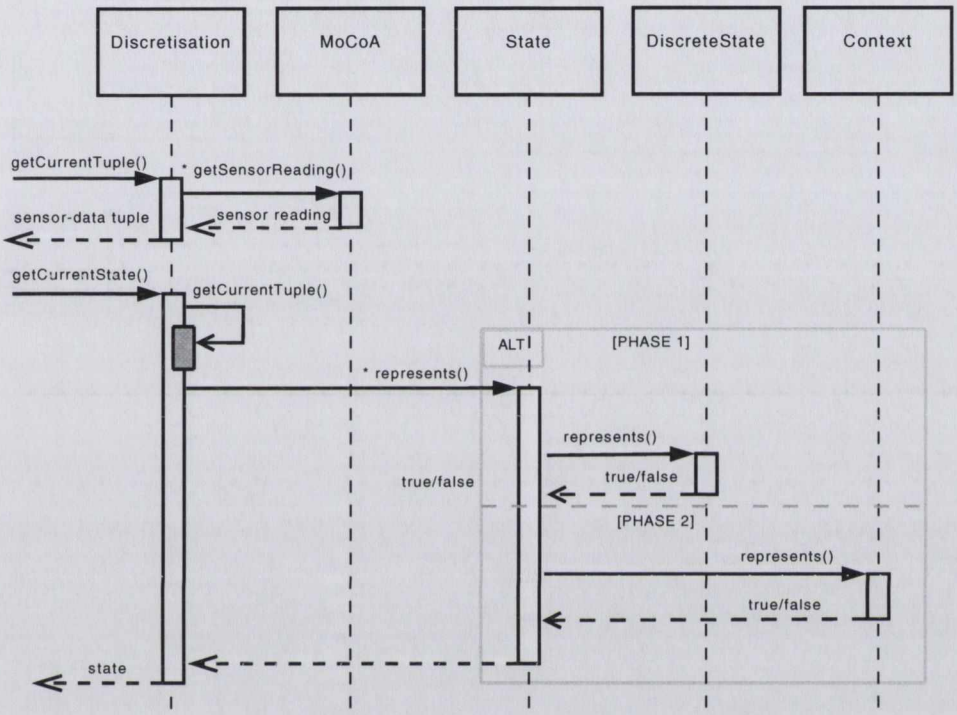

Figure 4.11: *Discretisation*– operations

4.5 Discretisation

The *Discretisation* class provides raw or discretised sensor data to other functional groups within KAFCA (Fig. 4.10). It manages the combination of sensor data into tuples and mapping tuples to *DiscreteStates* and *Contexts*. Fig. 4.11 illustrates the operations and interactions that occur when accessing sensor data through the *Discretisation* class. Raw sensor data is accessed through the *getCurrentTuple()* method. This method takes a vector of *Sensors* as a parameter and iteratively calls *getSensorReading()* on the *MoCoA* class to retrieve the current reading for each *Sensor*. These readings are combined and returned as a vector of doubles. The *getCurrentState()* method takes vectors of *Sensors* and *States* as parameters. It retrieves the current sensor-data tuples and then iterates through the *States* calling the *represents()* method to identify the representative *State* for that tuple. The *represents()* methods of the *DiscreteState* and *Context* classes implement the discretisation

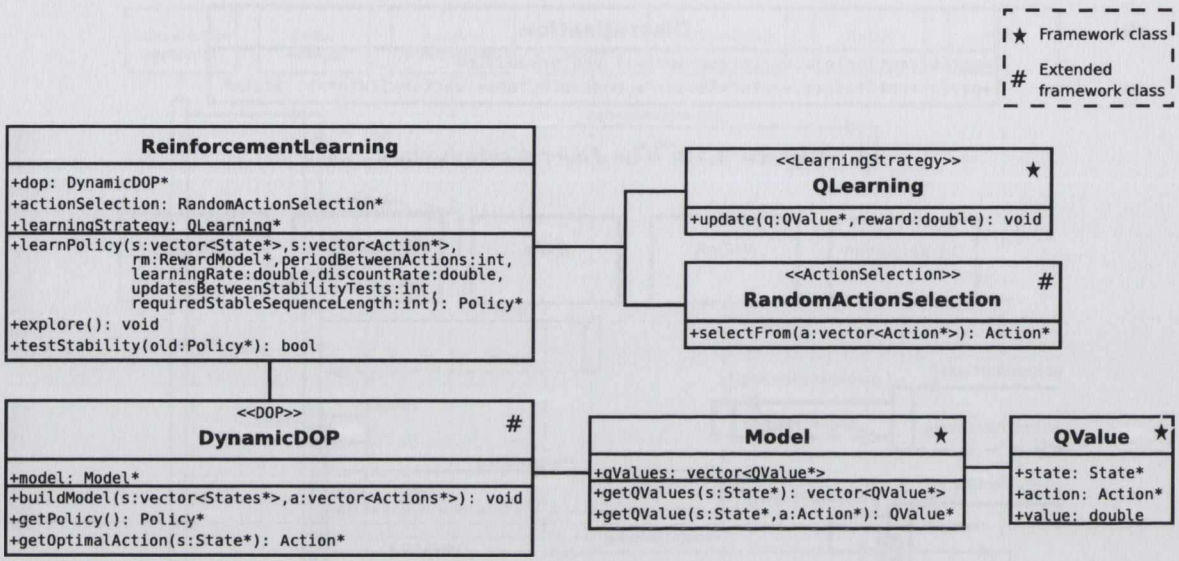

Figure 4.12: The reinforcement-learning classes

algorithms described in Section 3.3.

4.6 Reinforcement learning

The reinforcement-learning functional group learns policies that map *DiscreteStates* or *Contexts* to application *Actions*. Our implementation of this functionality extends the reinforcement-learning framework described in (Salkham et al., 2008). This framework provides commonly-used classes for reinforcement learning, and implementations of different action-selection and knowledge-update algorithms. However the framework does not support all of our requirements for reinforcement learning therefore we extend its classes to provide additional functionality.

Fig. 4.12 illustrates the classes used for reinforcement learning, and indicates classes that are from the existing framework and classes that have been extended. The *ReinforcementLearning* class provides an entry point to the functionality through its *learnPolicy()* method. This method takes the application *Actions*, *States*, *RewardModel* and configuration parameters for reinforcement learning, and returns a *Policy* that maps parameter *States* to *Actions*.

The *QValue* class encapsulates a *State* and *Action*, and the *value* of taking the action in the state. The *Model* class encapsulates a vector of *QValues* that represent all possible *State–Action* combinations. The discrete optimisation problem (*DOP*) class of the framework assumes that the set of states is fixed at design time by a developer, therefore we extend this class with the *DynamicDOP*

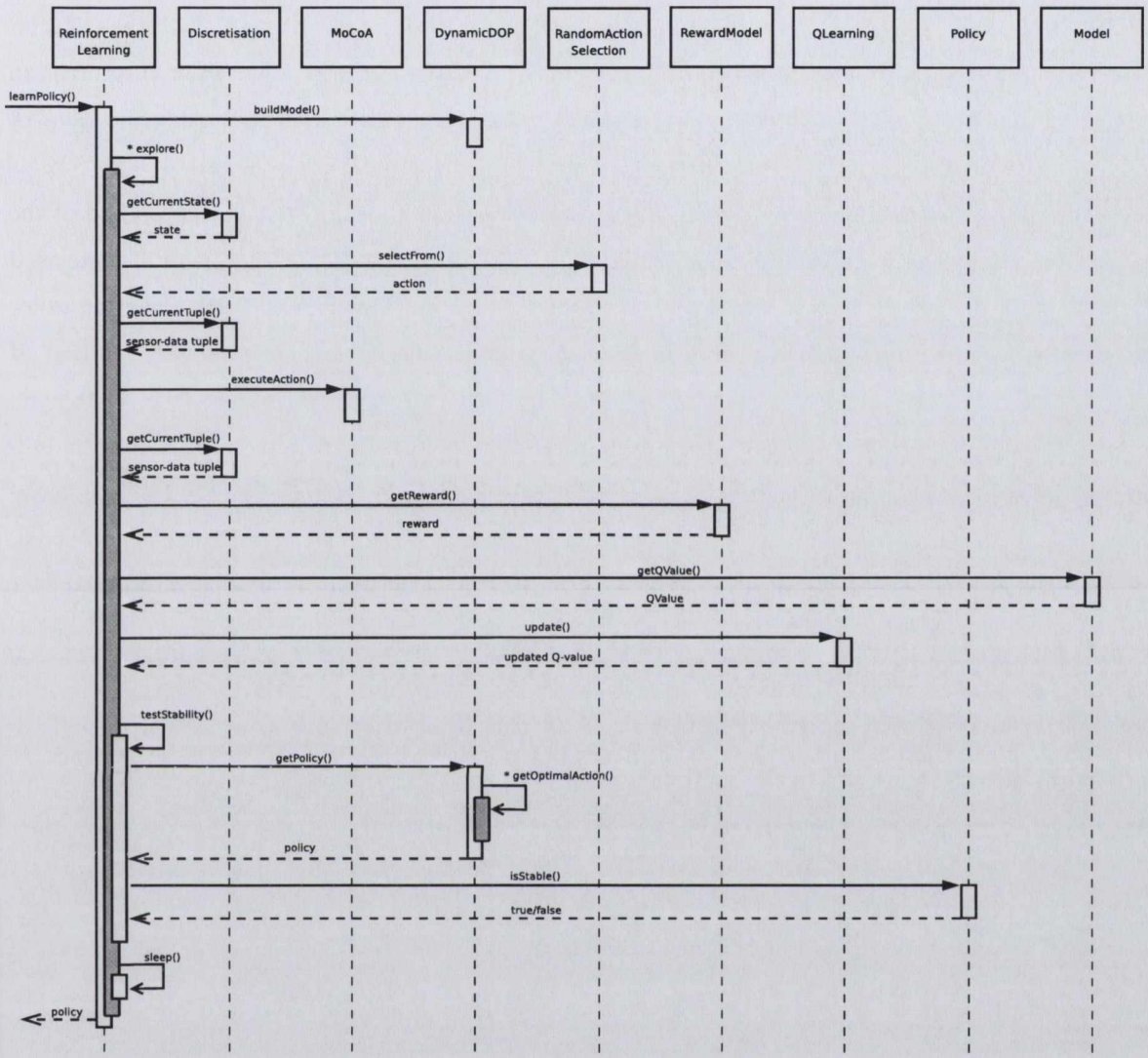

Figure 4.13: Reinforcement-learning– sequence of operations

class that provides functionality for constructing a *Model* at run time. The *QLearning* class implements the Q-learning update function (Alg. 6). The *RandomActionSelection* class extends the framework's abstract *ActionSelection* class and implements the *selectFrom()* method so that it selects an *Action* randomly from the parameter vector of *Actions*.

Fig. 4.13 illustrates the sequence of operations that occur when learning a *Policy*. The *learnPolicy()* method of the *ReinforcementLearning* class initiates reinforcement learning. Its first action is to create a *Model* of *QValue* instances for the parameter *States* and *Actions* by invoking *buildModel()* on the *DynamicDOP* class. The *explore()* method is then called every *periodBetweenActions* seconds

until a stable *Policy* is learned (Listing 4.2). The *explore()* method implements the five steps of reinforcement learning (Section 3.4). It first gets the current *State* from the *Discretisation* class. It then selects an action using the *RandomActionSelection* class. The action is executed using the *MoCoA* class and a reward is calculated by the application-specific *RewardModel* implementation. The *QValue* for the *State* and *Action* is retrieved from the *Model* and updated using the *update()* method of the *QLearning* class. Every *updatesBetweenStabilityTests* updates the *testStability()* method is invoked (Listing 4.3). The method creates a new *Policy* based on current Q-values and this *Policy* instance is compared to the *Policy* instance from the previous stability check using the *isStable()* method. If they are dissimilar then the new *Policy* replaces the old policy, i.e., the stable sequence is set to zero. It they are the same then the *stableSequence* attribute of the old Policy is incremented and the new *Policy* is discarded (it is a duplicate). The value of *stableSequence* is compared to the *requiredStableSequenceLength* parameter to test if a sufficiently long sequence of the same policy has been observed. When a sufficiently long stable sequence is observed learning finishes and the *Policy* is returned.

```
1  //Get the current state of the environment
2  State s = Discretisation->getCurrentState(sensors);
3  //Select a random action to execute
4  Action a = RandomActionSelection->selectFrom(actions);
5  //Get the sensor-data specified by the RewardModel
6  vector<double> preAction =
       Discretisation->getCurrentTuple(RewardModel->sensors);
7  //Execute the action
8  MOCOA->executeAction(a);
9  //Get the sensor-data after the Action
10 vector<double> postAction =
       Discretisation->getCurrentTuple(RewardModel->sensors);
11 //Calculate the reward
12 double reward = RewardModel->getReward(a, preAction, postAction);
13 //Select and update the Q-value
14 QValue q = Model->getQValue(s, a);
15 QLearning->update(q, reward);
16 numberOfUpdates++;
17 //If sufficient updates have occurred test the stability of the Policy
18 if (numberOfUpdates % updatesBetweenStabilityChecks == 0){
19   if (testStability() == true)
```

```
20      return DynamicDOP->getPolicy();
21  }
22  //Sleep for an application-specific period between actions
23  Sleep(periodBetweenActions);
```

Listing 4.2: The explore() method

```
1   //create a Policy from current QValues
2   Policy newPolicy = DynamicDOP->getPolicy();
3   //check if all state actions are the same in old and new policies
4   bool allElementsMatch = true;
5   for (unsigned int i=0; i<oldPolicy->elements.size(); i++){
6     PolicyElement pOld = oldPolicy->elements[i];
7     if (! pOld->a->equals(newPolicy->getActionForState(pOld->s))
8       allElementsMatch = false;
9   }
10  //if all elements match then increment the stable sequence counter
11  if (allElementsMatch)
12    oldPolicy->stableSequence++;
13  else
14    oldPolicy = newPolicy;
15  //if the observed sequence matches the required sequence
16  if (oldPolicy->isStable(requiredStableSequenceLength))
17    return true;
18  else
19    return false;
```

Listing 4.3: The testStability() method

4.7 Accurate context definition

The accurate-context-definition functional group applies the algorithms discussed in Section 3.5 to define accurate context definitions. Fig. 4.14 illustrates the *AccurateContextDefinition* class and Fig. 4.15 illustrates the sequence of operations for accurate context definition. The *define()* method of the *AccurateContextDefinition* class is the entry point to this functionality. The first step of the process is

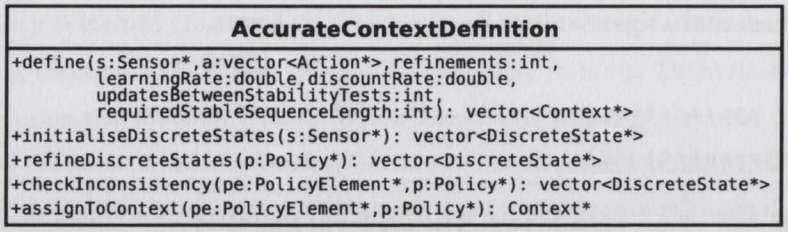

Figure 4.14: The *AccurateContextDefinition* class

to initialise discrete states for the parameter *Sensor* by invoking the *initialiseDiscreteStates()* method. This method implements the initialisation algorithm (Alg. 7) and creates *DiscreteState* instances that represent sensor values between the *Sensor's min* and *max* values.

The *refineDiscreteStates()* method is then called *refinements* times to refine the default *DiscreteState* instances of the *Sensor*. This method invokes *learnPolicy()* on the *ReinforcementLearning* class to learn a *Policy* for the current set of *DiscreteStates*, as described in Section 4.6. The *checkInconsistency()* method is then called for each *PolicyElement* to check if its encapsulated *DiscreteState* is inconsistent (Listing 4.4). Neighbouring *PolicyElements* are retrieved and compared to a *PolicyElement* using the *similarTo()* method. If the *PolicyElement* has dissimilar neighbours then the dimensions of the encapsulated *DiscreteState* must be split. New *DiscreteState* instances are created to represent the divided sensor space, as per the algorithm for splitting inconsistent discrete states (Alg. 8). A vector of refined *DiscreteStates* is returned from each iteration of the *refineDiscreteStates()* method.

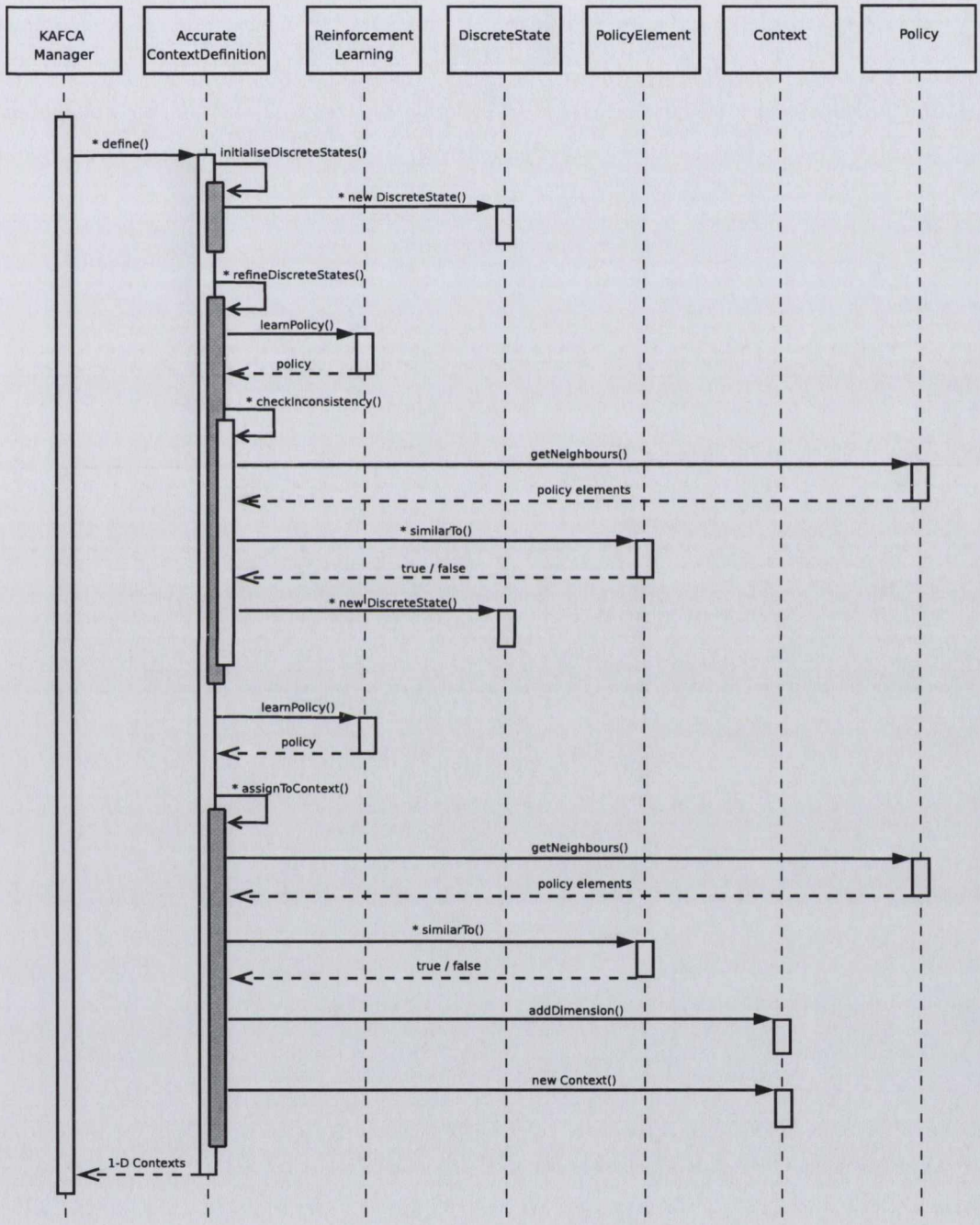

Figure 4.15: Accurate context-definition– sequence of operations

Once the final refinement of *DiscreteStates* has occurred the accurate-context-definition process moves on to context definition. A *Policy* for the final set of *DiscreteStates* is learned. The *assignToContext()* method is called for each *PolicyElement* to assign its encapsulated *DiscreteState* to a *Context* (Listing 4.5). This method implements the context-definition algorithm (Alg. 9). Neighbouring *PolicyElements* are retrieved from the *Policy* and compared to the parameter *PolicyElement*. Depending on their similarity and membership in existing *Contexts* the encapsulated *DiscreteState* is either added to an existing *Context* or a new *Context* is created.

After all *DiscreteStates* have been assigned to a *Context* the vector of one-dimensional *Contexts* for the parameter *Sensor* is returned.

```
1  //get the neighbours of the current Policy element being checked
2  vector<PolicyElement*> neighbours = Policy->getNeighbours(currentElement);
3  //create a record for dimensions in which the currentElement should be split
4  vector<bool> splitInDimension;
5  for (int i=0; i<currentElement->dimensions.size(); i++){
6    splitInDimension.push_back(false);
7  }
8  //mark dimensions in which there are dissimilar neighbours
9  for (int i=0; i<neighbours.size(); i++){
10   if (! currentElement->a->equals(neighbours[i]->a)){
11     //get the dimension in which they're neighbours
12     int d = currentElement->s->(shareEdgeInDimension(neighbours[i]->s));
13     splitInDimension[d] = true;
14   }
15 }
16 //refine the discrete state in dissimilar dimensions
17 DiscreteState s = (DiscreteState)currentElement->s;
18 vector<DiscreteState*> refinedStates;
19 refinedStates.push_back(s);
20 for (int i=0; i<splitInDimension.size(); i++){
21   //check if this dimension should be split
22   if (splitInDimension[i]){
23     vector<DiscreteState*> temp;
24     //each existing refined state must be split in two in dimension i
25     for (int j=0; j< refinedStates.size(); j++){
```

```
26        DiscreteState* s1 = new DiscreteState();
27        DiscreteState* s2 = new DiscreteState();
28        for (int k=0; k<refinedStates[j]->dimensions.size(); k++){
29          double lowerBound = refinedStates[j]->dimensions[k]->lowerBound;
30          double upperBound = refinedStates[j]->dimensions[k]->upperBound;
31          double midPoint = (upperBound - lowerBound) / 2;
32          //this is the dimension to be split
33          if (k == i){
34            s1->addDimension(new Range(lowerBound, midPoint));
35            s2->addDimension(new Range(midPoint, upperBound));
36          }
37          //these dimensions are unchanged by the current refinement
38          else {
39            s1->addDimension(new Range(lowerBound, upperBound));
40            s2->addDimension(new Range(lowerBound, upperBound));
41          }
42        }
43        //store in temp vector until existing refined states are processed
44        temp.push_back(s1);
45        temp.push_back(s2);
46      }
47      //assign newly refined states to this vector for more splitting
48      refinedStates.assign(temp.begin(), temp.end());
49    }
50  }
51  return refinedStates;
```

Listing 4.4: The checkInconsistency() method

```
1  //get the neighbours of the current Policy element being checked
2  vector<PolicyElement*> neighbours = Policy->getNeighbours(currentElement);
3  //check for similar neighbours and their Contexts
4  vector<DiscreteState*> similarNeighbours;
5  for (int i=0; i<neighbours.size(); i++){
6    if (currentElement->a->equals(neighbours[i]->a))
7      similarNeighbours->push_back(neighbours[i]->s);
```

```
8  }
9  //check if any similar neighbours are part of an existing Context
10 vector<Context*> similarNeighbourContexts;
11 for (int i=0; i<contexts.size(); i++){
12   for (int j=0; j<similarNeighbours.size(); j++){
13     if (contexts[i]->containsState(similarNeighbours[j]))
14       similarNeighbourContexts.push_back(contexts[i]);
15   }
16 }
17 //if neighbours not in existing context then create new context
18 if (similarNeighbourContexts.size()==0){
19   Context c = new Context();
20   c->dimensions.push_back(currentElement->s);
21   contexts.push_back(c);
22 }
23 //if one similar neighbour is in a context then add to that context
24 else if (similarNeighbourContexts.size()==1){
25   similarNeighbourContexts[0]->addDimension(currentElement->s);
26 }
27 //if more than one similar neighbour is in a context the combine the contexts
28 else if (similarNeighbourContexts.size()>1){
29   //assign the state to the first similar context
30   similarNeighbourContexts[0]->dimensions.push_back(currentElement->s);
31   //append the dimensions of other contexts to the first context
32   for (int i=1; i<similarNeighbourContexts.size(); i++){
33     similarNeighbourContexts[0]->dimensions.insert(
         similarNeighbourContexts[0]->dimensions.end(),
         similarNeighbourContexts[i]->dimensions.begin(),
         similarNeighbourContexts[i]->dimensions.end());
34     //erase the other context
35     vector<Context*>::iterator contextIterator;
36     contextIterator = contexts.begin();
37     while (contextIterator != contexts.end()){
38       if (*contextIterator == similarNeighbourContexts[i])
39         contexts.erase(contextIterator);
40       contextIterator++;
```

```
41        }
42    }
43 }
```

Listing 4.5: The `assignToContext()` method

4.8 Sensor selection

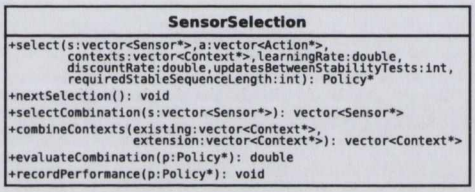

Figure 4.16: The *SensorSelection* class

The sensor-selection functional group applies the algorithms discussed in Section 3.6 to select the most suitable sensors for the application. Fig. 4.16 illustrates the *SensorSelection* class and Fig. 4.15 illustrates the sequence of operations for selecting sensors. The *select()* method of the *SensorSelection* class is the entry point to this functionality. This method is invoked once by the *KAFCAManager* to start the sensor-selection process. The *nextSelection()* method is then iteratively called until the best set of sensors is identified.

The first step of the *nextSelection()* method is to select a new sensor combination to evaluate using the *selectCombination()* method, which implements the combination-selection algorithm (Alg. 10). The *combineContexts()* method is then called to combine the *Contexts* of selected sensors (Listing 4.6). This method implements the context-combination algorithm (Alg. 11). A *Policy* for these n-dimensional *Contexts* is learned by invoking *learnPolicy()* on the *ReinforcementLearning* class. The sensor combination is evaluated by invoking *evaluateCombination()*. Within this method the *recordPerformance()* method (Listing 4.7) is invoked *numberOfEvaluationActions* times with *periodBetweenActions* seconds between each invocation. In each invocation the current *Context* is retrieved from the *Discretisation* class. The *Action* to take in this *Context* is retrieved from the *Policy* and executed by the *MoCoA* class. The performance of the application is recorded by the *update()* method of the *SensorEvaluationMetric* subclass specified by the application. Once all iterations of *recordPerformance()* have been carried out the *measurePerformance()* method is called on the *SensorEvaluationMetric* subclass. The result of this calculation feeds into the search for the best set of *Sensors*.

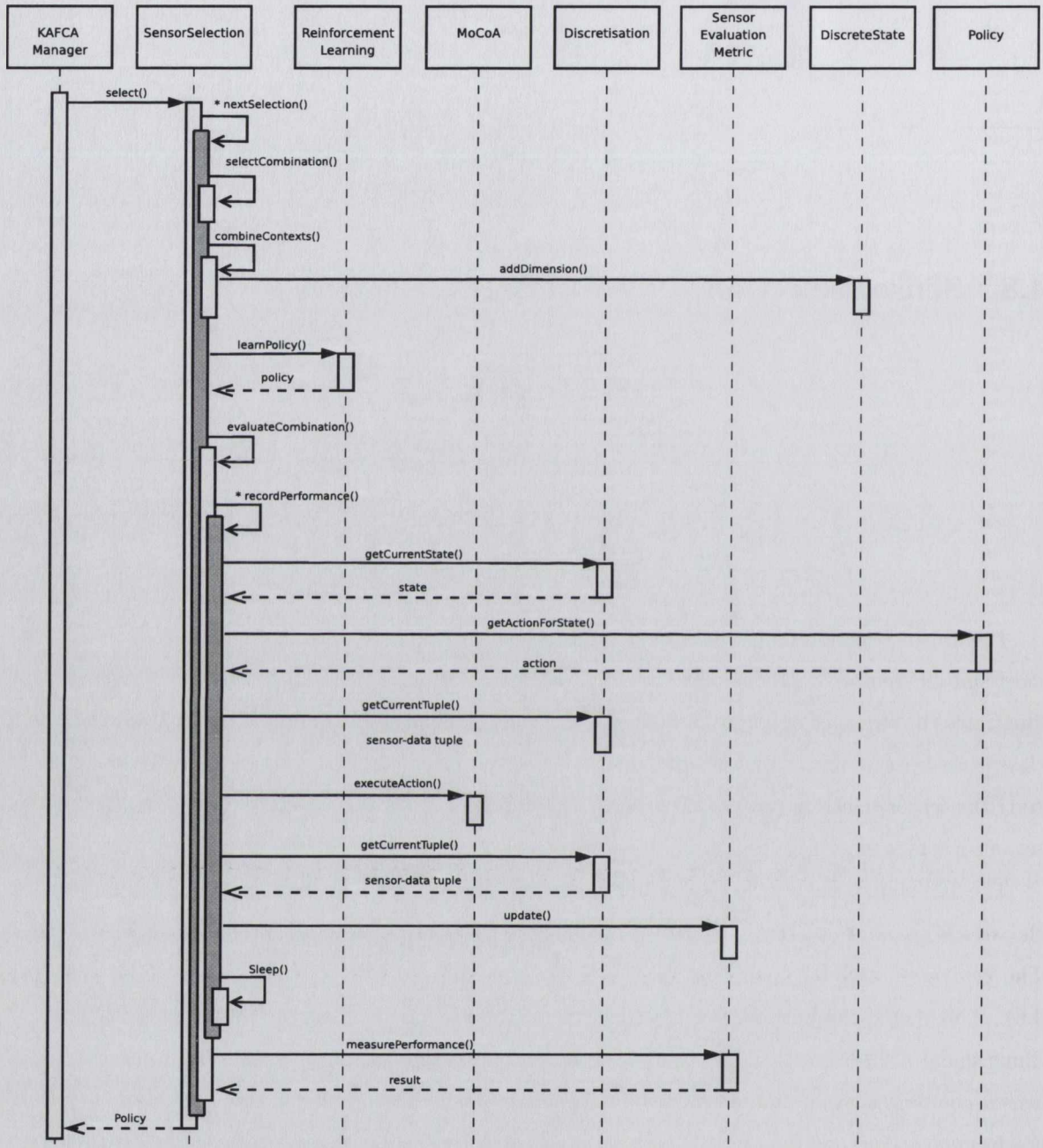

Figure 4.17: Sensor selection– sequence of operations

When the best-performing set of sensors is identified the *select()* method returns the *Policy* for that set of sensors.

```
1  //iterate through the contexts of the first sensor(s)
2  for (int i=0; i<sensor1Contexts.size(); i++){
3    //iterate through the discrete states of sensor 1's contexts
4    for (int j=0; j<sensor1Contexts[i]->dimensions.size(); j++){
5      //iterate through the contexts of the second sensor
6      for (int k=0; k<sensor2Contexts.size(); k++){
7        //iterate through the discrete states of sensor 2's contexts
8        for (int l=0; l<sensor2Contexts[k]->dimensions.size(); l++){
9          //extend each Discrete state of sensor 1
10         sensor1Contexts[i]->dimensions[j]->addDimension(
               sensor2Contexts[k]->dimensions[l]->dimensions[0];
11       }
12     }
13   }
14 }
```

Listing 4.6: The combineContexts() method

```
1  //Get the current state of the environment
2  State* st = discretisation->getCurrentState(sensors);
3  //Get the sensor data specified by the SensorEvaluationMetric
4  vector<double> preAction =
       discretisation->getCurrentTuple(SensorEvaluationMetric->sensors);
5  //Select and execute the appropriate action for this state
6  Action* a = policy->getActionForState(st);
7  MoCoA->executeAction(a);
8  //Get the sensor data specified by the SensorEvaluationMetric
9  vector<double> postAction =
       discretisation->getCurrentTuple(SensorEvaluationMetric->sensors);
10 //update the record of application performance
11 sensorEvaluationMetric->update(a, preAction, postAction);
12 //Sleep for an application-specific period between actions
13 Sleep(periodBetweenActions);
```

Listing 4.7: The recordPerformance() method

4.9 Chapter summary

This chapter described the C++ implementation of our approach to knowledge-autonomous context awareness. Initially we described a high-level overview of functional groups within the implementation, and the interactions that occur between these groups. This was followed by descriptions of the classes and their interactions within functional groups. We also discussed configuration details and implementation code for significant operations.

Chapter 5

Evaluation

This chapter describes the evaluation of our approach to knowledge autonomy for context-aware applications. We begin by outlining the goals of the evaluation and discuss each scenario relative to these goals.

5.1 Evaluation goals

In Chapter 1 we defined the objectives of the thesis: to accurately identify application contexts from sensor data interpreted at run time, and to select suitable sensors for identifying application contexts at run time. The goals of this evaluation are to measure how effectively each of these thesis objectives have been achieved. We also define an additional goal of the evaluation, which is to evaluate reinforcement learning as an appropriate technique for learning interpretations of discretised sensor data. We now describe how the implemented scenarios fulfill these goals.

KAFCA applies reinforcement learning to learn policies for discretised sensor data. It is difficult to follow the progress of the learning process in complex scenarios due to the stochastic nature of reinforcement learning and the significant number of learning iterations that are necessary. Our first two scenarios, *line* and *grid*, are simple scenarios that are intended to identify issues when applying reinforcement learning to learn policies for discretised sensor data. They apply reinforcement learning processes that use both types of reward model, immediate and long-term, and they address the goal of identifying if reinforcement learning is suitable for learning about sensor data.

The first phase of the KAFCA process addresses accurate run-time context definition, the first objective of the thesis. The *sentient-couch* scenario evaluates the effect of refining discrete states at run time on the accuracy of context definitions. The sentient couch is a sensor-enabled device that

identifies occupants based on their weight, and the application's task is to learn the context definitions that distinguish different occupants. The progression of context definitions is recorded over a number of refinements and compared to the known, ideal context definitions to measure their inaccuracy. The results indicate the effect of the first thesis objective on the context definitions of context-aware applications. A cost-benefit analysis of context-definition accuracy versus learning time is also carried out.

The second phase of the KAFCA process addresses the selection of suitable sensors at run time, the second objective of the thesis. The *power-management* scenario evaluates the effect of selecting sensors at run time on application performance. The application in this scenario manages the power status of a computer monitor. Experiments are based on both real-world sensor data gathered from users in a variety of office environments, and also on generated sensor data. Three different classes of context-aware applications are tested and compared to measure the relative effectiveness of sensor selection at run time– knowledge-intensive applications, learning applications that use fixed sets of sensors, and an application that selects sensors at run time using KAFCA. Application performance is measured using metrics for energy efficiency, user-perceived performance and device life expectancy. These metrics indicate the effect of the second thesis objective on context-aware applications.

5.2 Considerations in the selection of scenarios

A variety of reasons lead to the selection of the scenarios described in this chapter. Ideally there would be a set of benchmarking tests against which we could compare our approach however our review of the related approaches in Chapter 2 revealed no common set of tests or applications. In general, each approach is focused on a particular type of context-aware application, and the learning technique they apply is determined by the underlying sensor data for the type of application. For example, hidden Markov models are commonly used for processing sound data while data mining techniques are used where large collections of high-level examples are available. It is difficult to directly compare different, best-of-breed approaches as their suitability to particular applications is varied.

Another factor, and significant challenge, in selecting scenarios for this thesis was the availability of data sets of sensor data. Our basic requirements were that sensor data be in its raw form, and from multiple sensors that produce ordered sensor data. More particularly this sensor data needed to be gathered in various deployment environments to create the potential for identifying suitable sensors, and there needed to be a recorded source of environmental feedback from which a reinforcement-learning reward could be gathered. Ultimately these constraints dictated that only one scenario

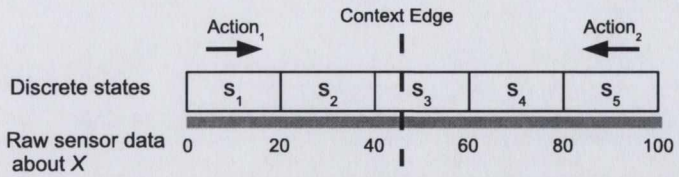

Figure 5.1: The *line* scenario

would be based on real sensor data, with the remaining scenarios simulating sensor data in some way.

5.3 The *line* scenario

The *line* scenario represents a one-dimensional sensor space that contains a single context edge. These characteristics make it the simplest possible scenario in which to evaluate reinforcement learning. The sole sensor for the *line* scenario measures a simulated characteristic X of the environment and produces data in the range 0-100. Its precision is 0.01 units and its initial discrete state boundaries are at 20, 40, 60 and 80 units (Fig. 5.1).

The application in this scenario affects the environment using two actions. $Action_1$ increases the value of X towards 100 and $Action_2$ decreases it towards 0. The goal of the application is to keep the value of X near to the context edge. This might equate to a temperature-control application that keeps the room temperature near a particular level. In order to achieve its goal the application should learn to select $Action_1$ for sensor values between 0 and the context edge, and select $Action_2$ for sensor values between the context edge and 100. The reward model reflects this. A positive reward (+1) is given when $Action_1$ is executed below the context edge and a negative reward (-1) when executed above it. $Action_2$ is similarly rewarded in the opposite cases.

5.3.0.1 Lessons

Our experiments with this scenario indicated that reinforcement learning could be an effective technique for learning policies that indicate how sensor data influences action selection. However we observed that the learned policy was not always consistent across different experiments, even when the context edge did not change. Our first reaction to this was to increase the number of learning iterations to ensure that sufficient learning occurred and Q-values had time to stabilise, however this solution was ineffective. To better understand the problem we recorded and examined the progress of Q-values during an experiment. We expected to observe a smooth learning curve as the optimal action emerged for each state but instead we observed that for one state (s_3) the Q-values oscillated. The

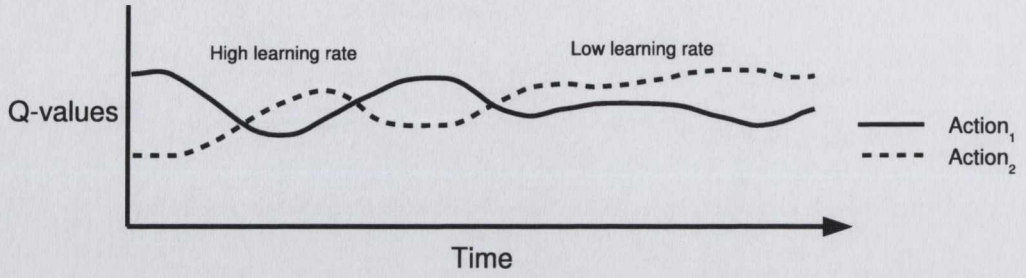

Figure 5.2: Q-value oscillation in an inconsistent discrete state

action for that state in the policy depended on the phase of the oscillation when learning stopped. We also observed that the extent of the oscillation depended on the size of the action effect, e.g., where the effect was small (a 1 unit change to the value of X) the oscillation was more pronounced than where the effect was larger (a 5 unit change to the value of X). No oscillation was observed when the effect of each action was set to a random amount.

The source of the oscillation became clear when we examined the Q-values and the value of X in parallel. The learned action for s_3 depended on the most recent part of its subspace to be visited during learning, i.e., there was a *spatial dependency* between the subspace and the Q-values. The rewards gathered to the left of the context edge within s_3 made Q-values oscillate such that $Action_1$ appeared optimal, and the rewards gathered to the right of the context edge within s_3 made Q-values oscillate such that $Action_2$ appeared optimal. This was our first encounter with an *inconsistent discrete state*, described in Section 3.4.3. The observed oscillation was more pronounced when the effect of each action on X was small, as it took more actions to move across the sensor space. The application would spend longer within each part of the inconsistent state, execute more actions, gather more rewards, and cause more Q-value oscillation.

Through further experimentation we discovered that the magnitude of the oscillation could be controlled by the *learning rate* used in the Q-value update function (Section 3.4.3). Fig. 5.2 exemplifies the oscillation that occurs in the Q-values for an inconsistent discrete state such as s_3. At high learning-rates a Q-value is mainly affected by the most recent updates. This makes the oscillation more pronounced as Q-values can change more quickly. Very low learning rates smooth the learning curve by reducing the effect of individual updates, so that Q-values represent updates over a longer period, i.e., across the entire subspace of an inconsistent state. Some oscillation still occurs due to the spatial dependency but it is insufficient to cause the perceived optimal action to change. Using a very-low learning rate (~ 0.01) we were able to learn consistent policies across different experiments.

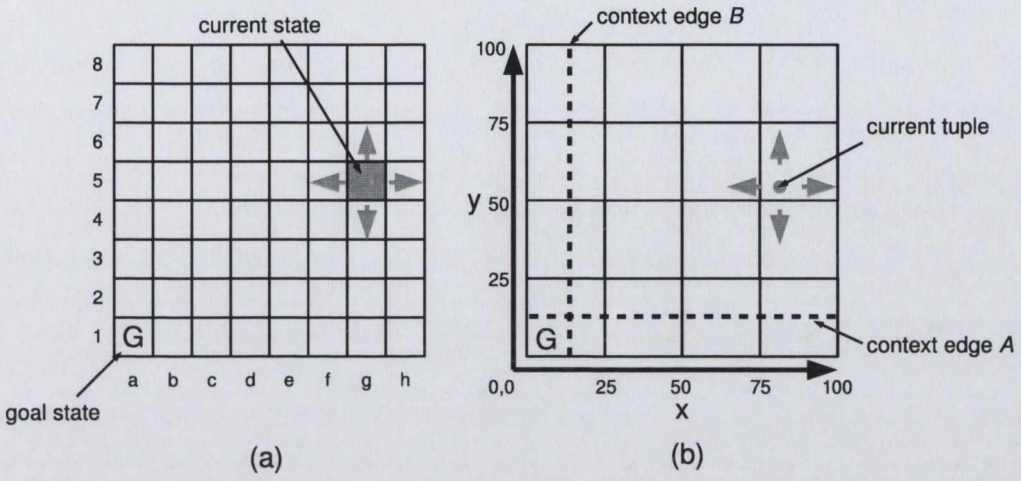

Figure 5.3: the *grid* scenario

5.4 The *grid* scenario

The *grid* scenario is a variation on a classic reinforcement-learning scenario (Sutton & Barto, 1998). In the classic scenario an application moves around a grid of states where one state is specified as a goal state (Fig. 5.3a). The application is aware of the state it currently occupies in the grid. The application's task is to learn a policy that guides it from any state in the grid to the goal state using *up*, *down*, *left* and *right* actions. These actions cause deterministic transitions between grid states in the classic scenario. The reward model gives a positive reward of 100 for any action that causes a transition to the goal state, and -1 for every other action.

This is an interesting scenario as it allows us to evaluate learning policies for discretised sensor data where the rewards are long-term. In our interpretation of this scenario the environment is a sensor space in two dimensions. The X and Y sensors in this scenario measure the coordinates of the application in the sensor space. They both produce data in the range 0–100, their precision is 0.01 units and they have initial discrete-state boundaries at 25, 50 and 75 units (Fig. 5.3b). We use the same long-term reward model as in the classic scenario however the "goal" is a particular subspace rather than a state.

5.4.0.2 Lessons

As in the *line* scenario we encountered difficulty learning consistent policies in the *grid* scenario. There were two causes of this inconsistency.

The first cause was that for a large region of the sensor space more than one action is optimal.

In Fig. 5.3b it is equally optimal to execute either *down* or *left* in the subspace above context edge A and to the right of context edge B. Both actions move the application equally towards the goal subspace. In an experiment the learned policy could map discrete states for this subspace to either *down* or *left*. From a reinforcement-learning perspective either is correct as both are optimal, however from the perspective of KAFCA it creates a potential issue when identifying the locations of context edges. If two neighbouring discrete states both have the same pair of equally optimal actions, and each state is associated with a different one of those actions, then it would appear that there is a context edge between two states when in reality there is none.

To address this problem we considered a solution where a policy could map a discrete state to more than one action. It is statistically very unlikely that both optimal actions would have identical Q-values, therefore there must be some threshold on what is considered "optimal", e.g., Q-values within 1% of the optimal Q-value are also considered optimal. This threshold would be configurable by the developer at design time. When we implemented this solution we observed that in policies some discrete states were accurately identified as having more than one optimal action, however we also observed discrete states where there was significant variation between Q-values of actions that should be equally optimal (10-15% in some cases). These variations were too wide to be reasonably considered "optimal" using our threshold-on-optimality solution.

By examining Q-values over the course of an experiment we established that these cases were caused by a different, second cause of inconsistency in learned policies. In the *line* scenario a spatial dependency exists between the point in the sensor space where an action is taken and the immediate reward for that action. The *grid* scenario uses a long-term reward model rather than an immediate one so such a spatial dependency does not exist. Instead a Q-value is influenced by the state transition that an action causes, as these transitions propagate long-term rewards from other states. These transitions are the second cause of inconsistency in learned policies for the *grid* scenario as learned Q-values depend on the order in which transitions occur, i.e., there is a *temporal dependency* between Q-values and transitions.

A long-term reward model creates a *reward path* from the goal to other discrete states. Fig. 5.4 illustrates how rewards are propagated through a reward path. Discrete states that do not transition directly to the goal subspace can only gather rewards via intermediary discrete states. s_1, s_2 and s_5 are the only discrete states that can transition directly to the goal subspace and receive the associated immediate reward. All other states must learn from rewards propagated from these states. When transition t_1 occurs an immediate reward is given to the action *left* in discrete state s_2. If transition t_2 occurred *before* t_1 then there would be no reward for t_2 to propagate, however if t_2 occurs *after* t_1

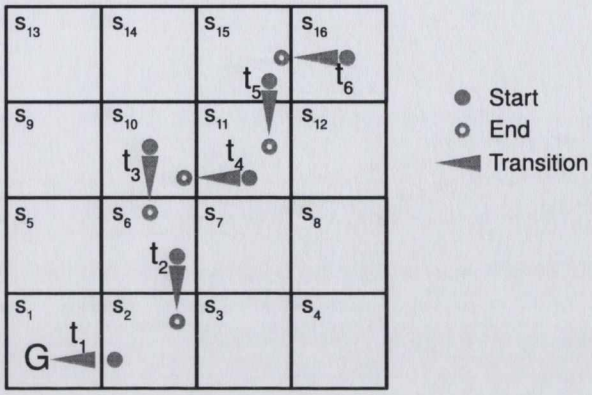

Figure 5.4: Temporal dependency in the *grid* scenario

then the transition propagates some of the immediate reward received at t_1. Similarly transition t_3 propagates some of the reward that was propagated by t_2. In Fig. 5.4 the transitions form a reward path from s_1 to s_{16} via six transitions. The final transition t_6 from s_{16} to s_{15} rewards the action *left* in s_{16} with some of the reward originally received for t_1. If only these transitions propagate rewards then the optimal action for s_{16} will be perceived to be *left*, even though *left* and *down* both lead equally optimally to the goal subspace. Obviously many more transitions and reward propagations occur during learning, however this example captures the temporal dependency in a reward path.

The reward path creates a temporal dependency as there is an order in which rewards propagate between discrete states. There may be many reward paths that lead from a discrete state to the goal. A discrete state's Q-values depend on the Q-values of other states in those paths. In our experiments the cases that we identified where there was significant variation between the Q-values of actions that should be equally optimal were a result of more transitions occurring on one path than on another. More rewards were propagated along one feedback path and therefore only one action appeared more optimal.

Through further experimentation we established that, similar to the spatial dependency, the temporal dependency could be addressed using a very low learning rate. This reduced the effect of individual updates (and transitions) on Q-values, so that rewards were propagated more slowly. The order in which transitions occurred was less of an issue as Q-values were learned over a longer learning period.

In this particular scenario the spatial dependency was more obvious due to the existence of equally-optimal actions in some regions of the sensor space. The success of our threshold-on-optimality approach for identifying equally optimal actions depended to a large extent on the threshold selected

by the developer, which would be an extra requirement for expert knowledge. We also intuitively felt that scenarios where multiple actions were optimal were unlikely to arise in real-world environments, as such environments are too non-deterministic for actions to be exactly, equally optimal. Due to these considerations we ultimately decided that facilitating multiple optimal actions was an unnecessary extension of our approach and as such is excluded from the current version of KAFCA.

5.5 Sentient-couch scenario

In the *sentient-couch* scenario a context-aware application uses sensor data to identify the user currently occupying a couch. The scenario is based on an old psychoanalyst's couch in a shared office in Trinity College, which is augmented with industrial-load sensors so that the weight of a person on the couch can be sensed. Load sensing is well suited for context identification as changes in weight distribution are indicative of movement and interaction in the physical environment (Schmidt et al., 2002). The sentient-couch platform has previously been used for a number of context-aware applications. The first implementation was a custom application that identified an individual based on their weight and issued a personalised greeting (Wolfe, 2003). The application was subsequently rebuilt using the Sentient Object Model as an evaluation of its ability to model and implement systems (Biegel, 2004).

In this scenario the application must identify users that do not always weigh the same amount. Even in the course of a day their weight varies based on what they have eaten or the clothes they are wearing. The challenge is to identify the range of sensor readings that indicate a context where a particular user is on the couch.

We apply KAFCA's accurate-context-definition process to this task. Rather than use the physical sentient couch and its sensors we simulated its operation. There are a number of reasons for simulating rather than experimenting with real sensor data. An objective of this scenario is to evaluate the trade off between learning iterations and the accuracy of context definitions. As the number of refinements increases so do the number of learning iterations, and running the experiments in real time would prohibit a significant analysis of this trade off. Another reason for choosing simulation over real-world deployment is the repeatability of experiments. By repeating experiments we can establish that consistent, accurate policies are being learned using reinforcement learning, and that the issues encountered in the *line* and *grid* scenarios do not reoccur.

5.5.1 Simulation

In the simulation each simulated user has a *weight range* that defines the range of weights that can be detected while they are on the couch. The *MoCoACouch* class extends the *MoCoA* class from our implementation (Section 4.3) so that it encapsulates attributes for the current occupant and the most recent action executed, so that feedback from the user can be simulated. The *getAllSensors()* method instantiates two sensors– *weightSensor* and *userFeedbackSensor*. *weightSensor* has a range between 0 and 100kg, a precision of 0.01 kg, and initial discrete-state boundaries at 10, 20, 30, 40, 50, 60, 70, 80 and 90kg. It is used by the application to measure the weight currently on the couch. The *userFeedbackSensor* is specified by the reward model as the source of feedback that is used to calculate rewards. The *getSensorReading()* method returns sensor data for the weight on the couch and simulated user feedback for actions. (Listing 5.1). The *executeAction()* method updates the most recently executed action so it can be used to calculate simulated user feedback by comparing the action to the current couch occupant.

The couch's context is simulated using Monte Carlo simulation, where the next couch context is generated from a discrete probability distribution. This approach is used to generate values randomly according to a defined distribution. The probability of no change to the current context is 0.9, the probability of the couch becoming vacant is 0.05, and the probability that someone new gets on the couch is 0.05. This is implemented in the *nextCouchContext()* method of the *MoCoACouch* class.

```
1  //Current user on couch
2  User* occupant;
3  //Most recently executed action
4  Action* lastAction;
5  //get available sensors
6  vector<Sensor*> getAllSensors(){
7    //instantiate the two application sensors, set their meta data, and return
         them
8  }
9  double getSensorReading(Sensor s){
10   //Simulate the weight of the occupant on the couch
11   if (s->id == "weightSensor"){
12     if (occupant==NULL)
13       return 0;
14     else {
```

```
15        double random = (double)rand() / RAND_MAX;
16        return occupant->min + random * (occupant->max - occupant->min);
17      }
18    //Simulate user feedback
19    } else if (s->id == "userFeedbackSensor"){
20      if (occupant->id == lastAction->command)
21        return 1;
22      else
23        return -1;
24    }
25  }
26  //Store the last action executed
27  void executeAction(Action* a){
28    lastAction = a;
29  }
30  //set the next couch state
31  void nextCouchContext(){
32    //couch activity {no change, user gets off, user gets on}
33    //select a random number between 0 and 1
34    double random = (double)rand() / RAND_MAX;
35    //p(no change) = 0.9
36    if (random<0.9){
37      //no change to couch
38    }
39    //p(user gets off) = 0.05
40    else if (random < 0.95){
41      occupant = NULL;
42    }
43    //p(user gets on) = 0.05
44    else {
45      User* randomUser = selectRandomUser();
46      occupant = randomUser;
47    }
48  }
```

Listing 5.1: The MoCoACouch class

5.5.2 Application implementation

The implementation of the application in this scenario extends and instantiates the classes within KAFCA as discussed in Section 4.2.

The application creates eleven instances of the *Action* class. Ten of the instances identify the users of the application, i.e., possible couch occupants. The *actuator* attribute of each instance is "display" and the *command* attribute is the "userID" (*user1–user10*). Obviously in the simulation there is no physical actuator, however the "display" could be a screen on which the userID is displayed. The final action instance is the action to take when the couch has no occupant. The *actuator* and *command* attributes in this instance are set to *NULL*.

The application also specifies a *CouchRewardModel* subclass of *RewardModel* (Listing 5.2). The *getReward()* method returns a negative reward (-1) when feedback from the user indicates that they were identified incorrectly, and no reward (0) otherwise.

```
1  //set the sensors needed for this reward model
2  sensors.push_back(new Sensor("userFeedbackSensor"));
3
4  //Calculate the reward based on user feedback
5  double getReward(Action* a, double[] preAction, double[] postAction){
6    //if the action did not identify the right user
7    if (postAction[0] == -1){
8         return -1;
9    }
10   //if the action identified the right user
11   else if (postAction[0] == 1){
12        return 0;
13   }
14 }
```

Listing 5.2: The `CouchRewardModel` class

5.5.3 Configuration

The application must configure the reinforcement learning and accurate-context-definition subprocesses of KAFCA, as discussed in Section 4.2. The configuration parameters are shown in Table 5.1.

Parameter	Value
learningRate	0.01
discountFactor	0
periodBetweenActions	30 seconds
updatesBetweenStabilityTests	200
requiredStableSequenceLength	10
numberOfRefinements	9

Table 5.1: Sentient-couch configuration

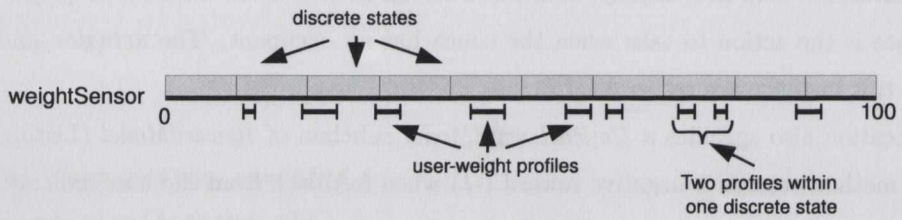

Figure 5.5: Example user-weight profiles

5.5.4 Experiment

Each experiment has 10 unique, randomly distributed user profiles. A random minimum weight is selected for each user. The maximum weight for a user is randomly selected in the range 1-5kg above their minimum weight (Fig. 5.5).

In each experiment context definitions are defined between each refinement of discrete states. The context edges of learned context definitions are compared to the known context definitions for simulated users. Each learned context definition has a lower and upper bound on the subspace it represents. These bounds should correspond to a user's minimum and maximum weight if the definition is accurate. The inaccuracy of each context definition (as a percentage) is the ratio of the context subspace that is inaccurate to the overall size of the context subspace.

$$\frac{[learnedLowerBound - userMinimum] + [learnedUpperBound - userMaximum]}{learnedUpperBound - learnedLowerBound} * 100$$

In each experiment the number of learning iterations required to learn a stable policy is also recorded for each refinement.

5.5.5 Results

The context-definition inaccuracy across 100 experiments is summarised in Fig. 5.6. The graph

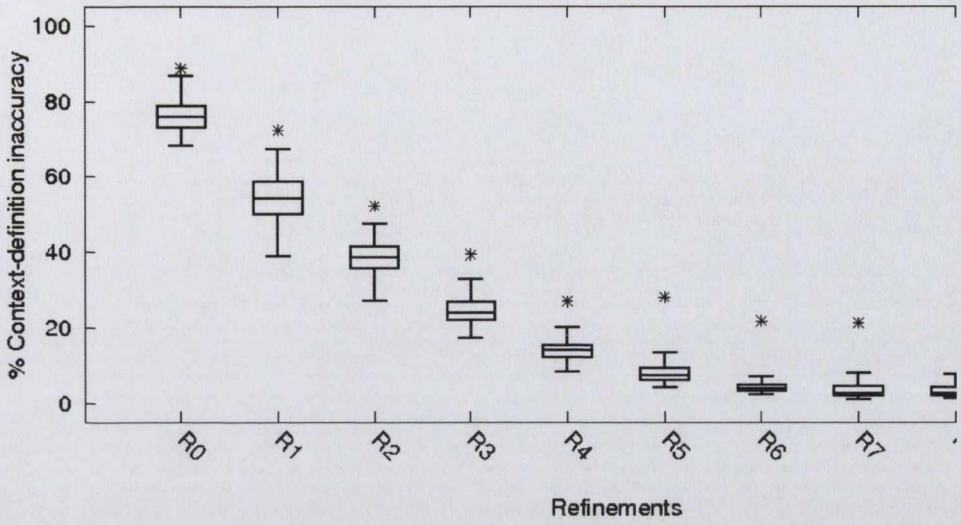

Figure 5.6: Context-definition inaccuracy

shows the progression in inaccuracy over 9 refinements of the KAFCA accurate-context-definition process, where $R0$ is the inaccuracy before any refinement occurs and $R1$ to $R9$ show the inaccuracy after each refinement. The bar-and-whisker plots (or box plots) show the median, interquartile range, valid range, and outliers in the data. The median is used rather than the mean as the mean is very sensitive to extreme observations, which may distort the view of what is typical (Baron, 2007).

The plot for $R0$ summarises the inaccuracy of context definitions based on initial discrete states. The median inaccuracy is 75.9%, with most values in the range 68.2–86.6%. At this stage of the experiment we observe that a number of contexts ($\sim$ 2 per experiment) go undetected. This occurs when a single discrete state encapsulates more than one context (Fig. 5.5). These contexts can only be distinguished from each other when the state is divided in subsequent refinements.

After one refinement of the discrete states ($R1$) the median inaccuracy drops to 54.3%, with range 39.0–67.2%. This is a 28% reduction in inaccuracy compared to $R0$. Inaccuracy continues to fall significantly over a number of refinements when compared to $R0$, with further median drops of 20%, 19%, 13%, 9% and 5%, at $R2$, $R3$, $R4$, $R5$ and $R6$ respectively.

By $R6$ the median inaccuracy is just 3.8%, a total drop of 95% from the inaccuracy at $R0$. Beyond this point the improvements in accuracy are negligible. In $R9$ the inaccuracy actually increases due to the "failure" of our stability-based approach to stopping learning. By this refinement the subspaces of some discrete states are so small that learning is an extremely slow process. Each discrete state must be visited a number of times, and the probability of visiting tiny states is proportionally small. In some cases it appeared that the policy was stable when in fact learning was just extremely slow.

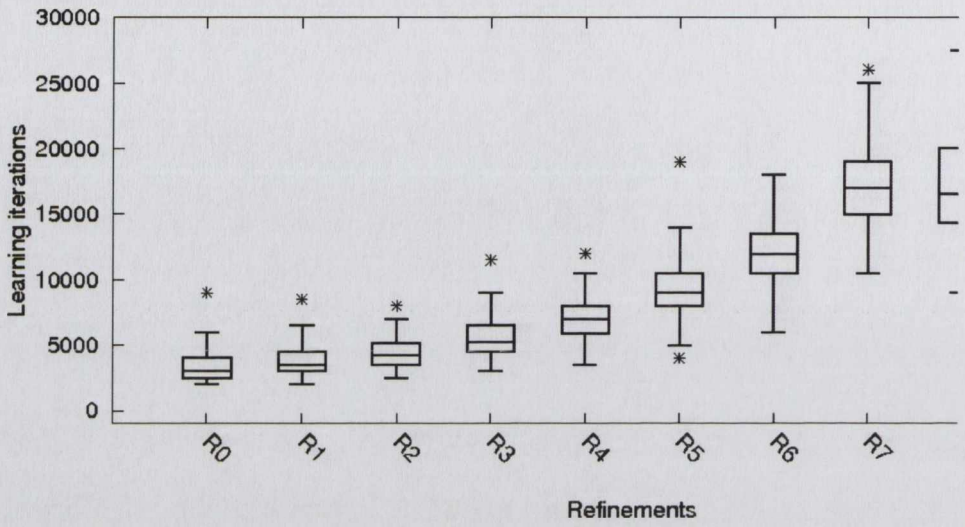

Figure 5.7: Required learning iterations

Learning was stopped prematurely and this caused learned policies to be inaccurate. As a result of inaccuracy in policies the learned context definitions were also inaccurate, and the median and range of inaccuracies actually increased as a result. This issue could be addressed by increasing the required sequence of stable policies and the updates between stability tests, however this demonstrates that ultimately there is no way to identify when sufficient learning has occurred. Our stability-based approach is merely a guide to identify when learning *appears* to be finished.

The number of required learning iterations was also recorded across the set of 100 experiments (Fig. 5.7). It should be noted that the minimum number of iterations to learn a stable policy is 2000, as we configure the application to stop learning after observing a sequence of the same policy 10 times, and this stability check is applied every 200 iterations. In fact the stability approach potentially increases the number of required iterations by 2000, as the first policy in the stable sequence is the same as the last, i.e., no further learning was necessary after the first policy in the stable sequence.

In $R0$ the median number of iterations is 3000, which suggests that approximately 1000 iterations were required to learn a stable policy (plus 2000 to establish that it was stable). In $R1$ the median is 3500, suggesting that 1500 iterations were required to learn a stable policy– a 50% increase over $R0$. As the number of refinements increases so does the required number of iterations. The median peaks at 17000 iterations in $R7$.

When we remove the iterations used to identify that the policy is stable we observe that each

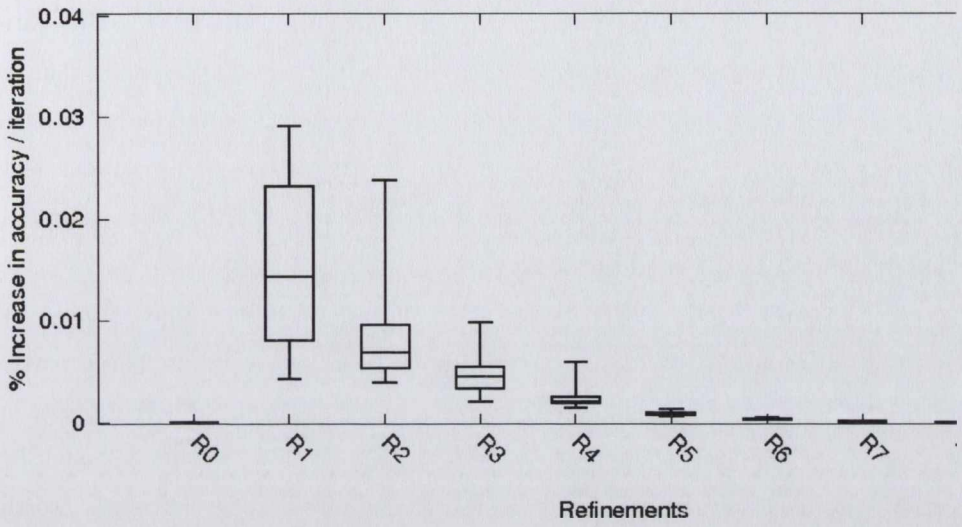

Figure 5.8: Cost-benefit analysis of discrete-state refinement

additional refinement between $R0$ and $R7$ requires on average 47% more iterations than the previous refinement (range 43–54%). This is not the case in refinements $R8$ and $R9$ where there is no appreciable increase over the iterations required in $R7$. This appears to be a result of the situation noted above, where learning was so slow that the policy appeared to be stable. If a longer *requiredStableSequenceLength* was configured then it is likely that we would observe the same rate of increase in required learning iterations as in other refinements.

A cost-benefit analysis of accuracy versus required learning iterations is shown in Fig. 5.8. The graph shows the % increase in context-definition accuracy per learning iteration for each refinement of the discrete states, compared to definitions for the previous version of the discrete states. The benefit at $R0$ is obviously 0 as no refinement has occurred. $R1$ yields a median 0.014% increase in context-definition accuracy over the context definitions at $R0$. The median yield drops off quickly as the number of refinements increases. $R2$, $R3$, $R4$ and $R5$ yield 0.006, 0.004, 0.002 and 0.001% respectively per learning iteration. $R6$–$R9$ yield almost no increase per learning iteration.

5.5.6 Conclusion

The first objective of the thesis is to accurately interpret application contexts at run time. These results show that KAFCA can increase the accuracy of context definitions at run time by refining discrete states. Across six refinements the median inaccuracy is reduced by 95%. The majority of

this improvement occurs by the third refinement (68%). A side-effect of refining the discrete-state subspaces is a requirement for increased numbers of learning iterations. Our analysis shows that in this scenario the required number of iterations increases by ~47% per refinement. This affects the benefit of each additional refinement– the first refinement yields more than 10 times the improvement in context-definition accuracy compared to the fifth refinement, which suggests that the number of refinements to apply in a scenario should be carefully chosen.

This thesis is concerned with the accuracy of context definitions, however it cannot be ignored that a very large number of learning iterations may be necessary with reinforcement learning. This is partly a consequence of the low learning rate we use to overcome the spatial dependency. Another factor in this scenario is the large number of possible application actions. Each of the eleven actions must be executed multiple times in each discrete state in order to learn an accurate policy. It is also worth noting that 50% of the time the couch is unoccupied based on our discrete-probability distribution (Section 5.5.1). Learning iterations when the couch is unoccupied do not provide the application with any knowledge about actions in user contexts.

These factors combine to increase the required number of learning iterations. In the real world it would be unrealistic to get feedback from a user for this magnitude of iterations. A reward model where the application does not depend on explicit user feedback, and instead senses feedback from the environment, would address this issue.

5.6 Power-management scenario

In the *power-management* scenario a context-aware application transparently manages the state of users' desktop monitors. The objective is to minimise the electricity consumption of the monitor while maintaining user-perceived performance. The application uses a variety of sensors to identify the context, and real-world sensor data for these sensors is used from a previous study (Harris, 2007). In this scenario we emulate the real world using this recorded sensor data.

The original study evaluated the potential energy savings of context-aware power management over traditional power-management techniques. A population of twenty office employees was selected for the study. Employees were chosen to represent a diverse set of users, with different computer-usage profiles and working in different office environments. They held a variety of jobs and worked in both shared and single offices. Their usage of the computer as well as sensor data for a set of sensors was recorded for each user, at five-second intervals, over a five-day period. The user's *usage trace* was recorded by observing idle periods when the keyboard and mouse were not in use (Fig. 5.9).

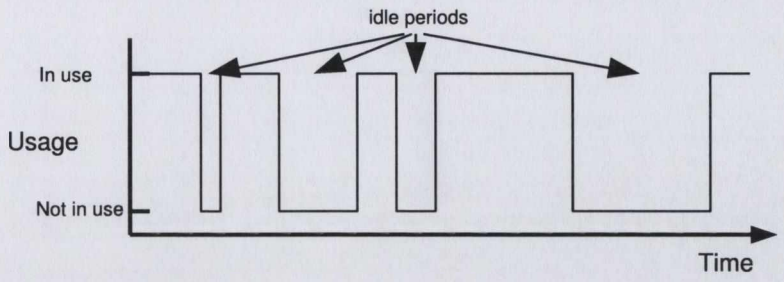

Figure 5.9: Recorded usage trace based on idle periods

5.6.1 Sensors

The sensors in the original study were chosen to detect the user at different distances from the computer. A bluetooth sensor is used to detect the user when they are anywhere in the office, by detecting a bluetooth dongle that the user carries. A microphone is similarly used to sense the user's presence anywhere in the office. A camera and ultra-sonic range finder are used to detect the user when they are at their desk. The idle time of the computer is modeled as a software sensor, which detects that the user is actually using the computer. It provides data concerning how long the computer has been idle.

The recorded sensor data was preprocessed for some sensors in the original study. There were two layers of processing, the first of which discretised some of the raw sensor data. The image produced by the camera was processed using a standard face-detection algorithm (Intel, 2006) so that it produced a binary output– either a face was detected or not. The microphone output was processed using a voice-activity algorithm (Intel, 2007) so that it produced an output in the range 0–100. Values represent the percentage of voice activity in the previous 5 seconds. Henceforth we refer to these as the *face-detection* and *voice-activity* sensors as these are the characteristics of the environment that they measure.

Sensor Value	Meaning
0	The user was detected in last period
1	One period since the user was detected
2	Two periods since ...
...	...

Table 5.2: Processed sensor data captures time

The second layer of processing added a time element to data from the *bluetooth* and *face-detection* sensors. Table 5.2 demonstrates how time was captured in a sensor value, by adding the notion of

Sensor	Meaning	Range	Precision	Initial boundaries
Bluetooth	Periods since tag detected	0 - ∞	1	1, 5, 10
Face detection	Periods since face detected	0 - ∞	1	1, 5, 10
Voice activity	% Voice activity in last period	0 - 100	1	25, 50, 75
Object range	Range (cm) to object	0 - 200	1	50, 100, 150
Idle time	Seconds since computer activity	0 - ∞	1	1, 60, 120, 300, 600
Power status	Monitor power state (suspended/active)	0, 1	0.5	0.5

Table 5.3: Power-management sensors

sequence to the data. Sensor data from the *object-range* and *idle-time* sensors was not preprocessed.

In addition to these sensors we define a *power-status* sensor that senses when the monitor is active or suspended. Table 5.3 summarises the data produced by each sensor in this scenario, as well as their meta data. The power-status sensor's precision and initial boundaries are set to 0.5. This means that only two discrete states are defined for it during discrete-state initialisation. Each of these represents one of its two possible values– 0 and 1.

All of these sensors fulfill our ordering requirement for sensor data (Section 3.1.2.2). This would not be the case without the preprocessing carried out on some of the sensor data, e.g., the image produced by the camera would not meet this requirement in its raw state. However preprocessing also causes data to be lost/hidden from the application. For example, the microphone is capable of detecting more than voice activity, e.g., it could detect a door being closed. Similarly the camera could detect more than the user's face, e.g., it could detect movement within the room. These potentially useful characteristics of the environment were lost during preprocessing as the original study focused on a specific set of characteristics. This motivates the provision of as much environmental detail as possible, and facilitating the application to autonomously choose what is useful.

5.6.2 Application goal

The fundamental assumptions of power management are that idle periods occur during the operation of a device and that these periods can be predicted with some certainty (Benini et al., 2000). Energy consumption is reduced by transitioning from higher to lower *power states* during idle periods. The challenge to power management is that most power state transitions have a significant cost. Typically they may:

1. Consume extra energy while making the transition.
2. Reduce the lifetime of the device. Some devices wear out faster when switched on and off frequently.

3. Reduce device performance from the user's perspective, e.g., the user may have to wait for the device to resume.

The first two concerns mean that not all idle periods are long enough to justify suspending a device. A power-management application should predict whether the idle period will be long enough to justify the transition cost (the break-even time). The third concern means that the application should also predict when the idle period will end, and power up the device so it is ready for the user.

We emulate power managing the same device as the original scenario, a computer monitor. The monitor consumes 45.8 Wh[1] while activated, and 1.8 Wh while suspended. The break-even time is 60 seconds, and the suspend and activate times are both 2 seconds (Harris, 2007). A power-management application must take these characteristics into account.

5.6.3 Emulation

In order to emulate the retrieval of sensor data from the real world the *MoCoAPowerManagement* class extends the *MoCoA* class. It encapsulates an attribute for the monitor's power state, and the *executeAction()* method changes the state of this attribute to emulate the monitor being activated or suspended. The *getAllSensors()* method instantiates and returns the six sensors in Table 5.3. The *getSensorReading()* method returns sensor data for a sensor from log files of recorded sensor data. Each entry in the log files is a timestamp–value pair. The emulated time is maintained inside the *MoCoAPowerManagement* class so that the correct sensor readings can be retrieved from log files based on their timestamps. If the end of the recorded sensor data is reached before a stable policy is learned then the emulation returns to the beginning of the recorded sensor data and the learning process continues.

```
1 //Power status: active = 1, suspended = 0
2 string devicePowerState = 1;
3 //Emulated time
4 long emulatedTime = -1;
5 //get available sensors
6 vector<Sensor*> getAllSensors(){
7   //instantiate the six application sensors, set their meta data, and return
        them
8 }
```

[1] 1 Watt hour (Wh) is the amount of energy needed to run a 1 Watt device for 1 hour

```
9  //retrieve the sensor reading from an emulated sensor
10 double getSensorReading(Sensor s){
11   //special case if the sensor detects the emulated power state
12   if (s->id == "devicePowerStateSensor")
13     return devicePowerState;
14   //initialise the emulated time to the start of the experiment
15   if (emulatedTime == -1){
16     //initialise to first timestamp in file;
17     emulatedTime = fileManager->getFirstTimestamp(s->id);
18   }
19   //get the value for the parameterised sensor at the current emulated time
20   double value = fileManager->getData(s->id, emulatedTime);
21   //check if reached end of file, make recursive call to get first reading
22   if (fileManager->endOfFile(s->id)){
23     emulatedTime == -1;
24     return getSensorReading(s);
25   }
26   //increment emulatedTime by the configured period between actions
27   else {
28     emulatedTime += periodBetweenActions;
29     return value;
30   }
31 }
32 //Change the power state of the device
33 void executeAction(Action* a){
34   if (a->command=="activate")
35     devicePowerState = 1;
36   else if (a->command=="suspend")
37     devicePowerState = 0;
38   //the defer action makes no change to the power state
39 }
```

Listing 5.3: The MoCoAPowerManagement class

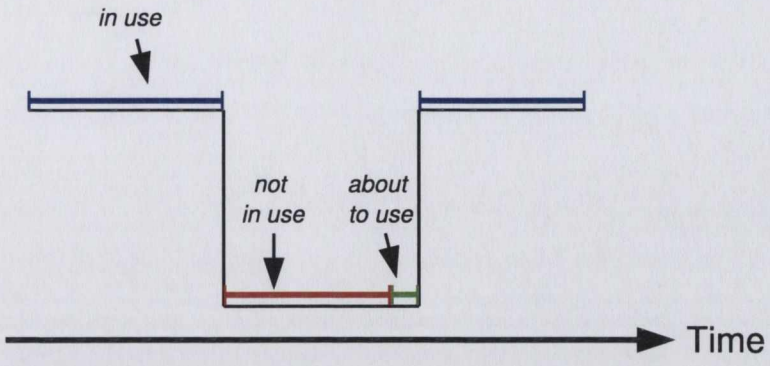

Figure 5.10: Key periods for changing the power state

5.6.4 Application implementation

The application in this scenario extends and instantiates KAFCA classes as discussed in Section 4.2. It also creates three instances of the *Action* class to control the power state of the monitor. The *suspend* and *activate* actions have obvious effects on the power state, and the *defer* action makes no change to the current power state.

5.6.4.1 Reward Model

The application also specifies a *PowerManagementRewardModel* subclass of *RewardModel*. This reward model went through a number of different iterations during experimentation. Our initial version took into account the amount of wasted idle period before suspending, as well as penalties for suspending when the monitor was still in use and also for not activating when the user was about to use the monitor. This reward model required that weights be set on these different elements so that an overall reward could be calculated. We experienced some initial success with this approach but found that it was necessary to tailor the weights to individual users to learn accurate policies. The correct weights for a user could only be identified through trial and error, and this depended on the developer identifying when it looked like "good" policies were being learned. Therefore we felt this model was too dependent on expert knowledge.

The second version of the reward model attempted to minimise the dependency on expert knowledge by simplifying the reward calculation. We identified two key periods where it was appropriate to change the power state of the monitor: when the computer is *not in use* it is appropriate to *suspend*, and when the user is *about to use* the computer it is appropriate to *activate* (Fig. 5.10). When the computer is *in use* it is appropriate to *defer* changing the power state. This version of the reward

model gave balanced rewards (+1 and -1) for selecting actions correctly or incorrectly in these periods. Our intuition was that the sensor data that was observed in these periods would be consistent and therefore would become correctly associated the appropriate actions for the different periods. For example, if the value 0 from the bluetooth sensor was observed in the *about-to-use* period but not in the *not-in-use* period then it would become associated with the *activate* action. However our subsequent analysis of the sensor data (Section 5.6.8) revealed that there was insufficient consistency in the sensor data and as a result this reward model was not effective.

The current version of the reward model builds on the approach used in the second version, however instead of using balanced rewards (+1 and -1) the rewards emphasise the importance of true positives over false positives, e.g., it is more important to *activate* correctly in the *about-to-use* period (+10) than it is to *activate* incorrectly in the *not-in-use* period (-1). This was the most effective reward model we identified during experimentation (Listing 5.4).

```
1  //set the sensors needed for this reward model
2  sensors.push_back(new Sensor("idleTimeSensor"));
3  sensors.push_back(new Sensor("devicePowerStateSensor"));
4
5  //Calculate the reward based on the idle time
6  double getReward(Action* a, double[] preAction, double[] postAction){
7    //check if there was an idle period after the action
8    bool idleAfterAction = (postAction[0] == 0);
9    //check if the device was active before the action
10   bool deviceActiveBeforeAction = (preAction[1] == 1);
11
12   if (a->command=="suspend"){
13     //suspended correctly if there was an idle period after suspense
14     if (idleAfterAction)
15       return 1;
16     else
17       return -2;
18   }
19   else if (a->command=="activate"){
20     //activated correctly if there was no idle period after activation
21     if (!idleAfterAction)
22       return 10;
```

```
23      else
24        return -1;
25    }
26    else if (a->command=="defer"){
27      //if its deferring suspend
28      if (deviceActiveBeforeAction){
29        //deferred suspend incorrectly as there was an idle period
30        if (idleAfterAction)
31          return -2;
32        else
33          return 1;
34      }
35      //if its deferring activation
36      else if (!deviceActiveBeforeAction){
37        //deferred activate correctly as there was an idle period
38        if (idleAfterAction)
39          return 10;
40        else
41          return -1;
42      }
43    }
44  }
```

Listing 5.4: The `PowerManagementRewardModel` class

5.6.4.2 Sensor-evaluation metric

The application also specifies a *PowerManagementSensorEvaluationMetric* subclass of *SensorEvaluationMetric*. This class implements the *update()* and *calculateResult()* methods using application-specific logic to measure application performance. Similar to the reward model this metric went through a number of iterations. The initial metric measured how optimally the application managed energy consumption by recording the proportion of idle periods that the monitor was suspended for. However during experimentation the sensors selected while using this metric did not provide good user-perceived performance. In particular the user was frequently forced to activate the computer manually at the end of the idle period. In fact this metric showed improved performance when the monitor was not activated automatically, as energy was saved in the *about-to-use* period where the monitor should

have been activated. This metric focused too much on the application's energy performance and did not capture all elements of the application's goals.

The current version of the metric measures application performance based on the ratio of correct action selections to the total number of actions taken (Listing. 5.5). The *update()* method records the occurrences of specific events, such as where the application failed to suspend the monitor when it was idle, or when the user was forced to manually power up the monitor. This version of the metric attempts to balance the importance of user-perceived performance and energy saving.

```
1  //set the sensors needed for this sensor-evaluation metric
2  sensors.push_back(new Sensor("idleTimeSensor"));
3  sensors.push_back(new Sensor("devicePowerStateSensor"));
4
5  //record occurrances of different events
6  int totalActivates = 0;
7  int totalSuspends = 0;
8  int manualActivates = 0;
9  int falseSuspends = 0;
10 int inefficientActivates = 0;
11 int failureToSuspends = 0;
12
13 //update the record of application behaviour
14 void update(Action* a, double[] preAction, double[] postAction){
15   //check if there was an idle period before and after the action
16   bool idleBeforeAction = (preAction[0] == 0);
17   bool idleAfterAction = (postAction[0] == 0);
18   //check if the device was active before the action
19   bool deviceActiveBeforeAction = (preAction[1] == 1);
20
21   //a general record of actions
22   if (a->command=="activate"){
23     totalActivates++;
24   }
25   else if (a->command="suspend"){
26     totalSuspends++;
27   }
```

```
28
29    //a manual activate occurs if the idle period ends but activate is not
          called
30    if (idleBeforeAction && !idleAfterAction && !a->command=="activate"){
31      manualActivates++;
32    }
33    //a false suspend occurs if there is no idle period after suspend is called
34    if (!idleAfterAction && a->command=="suspend"){
35      falseSuspends++;
36      //also causes a manual activate
37      manualActivates++;
38    }
39    //an inefficient activate occurs if activation is called before the end of
          the idle period
40    if (idleAfterAction && a->command=="activate"){
41      inefficientActivates++;
42    }
43    //a failure to suspend occurs if suspend is deferred in an idle period
44    if (idleAfterAction && deviceActiveBeforeAction &&
          action->command=="defer"){
45      failureToSuspends++;
46    }
47  }
48
49  double calculateResult(){
50    //if no contexts are detected by these sensors
51    if ((totalActivates + totalSuspends) == 0)
52      return -1;
53
54    //calculate the proportion of good activates
55    double goodActivateProportion = (totalActivates - manualActivates -
          inefficientActivates)/totalActivates;
56    //calculate the proportion of good suspends
57    double goodSuspendProportion = (totalSuspends - falseSuspends -
          failureToSuspends)/totalSuspends;
58    //return the average
```

Parameter	Value
learningRate	0.01
discountFactor	0
periodBetweenActions	5 seconds
updatesBetweenStabilityTests	1000
requiredStableSequenceLength	10
numberOfRefinements	4
numberOfEvaluationActions	5000

Table 5.4: Sentient-couch configuration

```
59    return (goodActivateProportion + goodSuspendProportion)/2;
60  }
```

Listing 5.5: The `PowerManagementSensorEvaluationMetric` class

5.6.5 Configuration

The application must configure KAFCA as discussed in Section 4.2. The configuration parameters are shown in Table 5.4.

5.6.6 Experiment

Each experiment is centred around the data for a particular user. The performance of different power-management applications are evaluated and compared during an experiment. A theoretical *Oracle* application is used to measure the optimal power consumption for a user. An *always-on* application is used to show the opposite end of the energy-efficiency spectrum. A number of *threshold* applications are also defined that represent standard, knowledge-intensive approaches to power management. KAFCA is used to learn context definitions for a set of context-aware applications that use fixed sensors. It is also used to learn context definitions for an application that selects the most suitable sensors at run time. The performance of each of these applications is measured and compared using a number of application-specific metrics. We now look at each of these applications and the evaluation metrics in more detail.

5.6.6.1 Oracle and always-on applications

The *Oracle* application is a theoretical application that has future knowledge of user requests for a device (Simunic et al., 2000). This application suspends the device at the beginning of an idle period that exceeds the break-even time. If the device is in the suspended state the application activates it

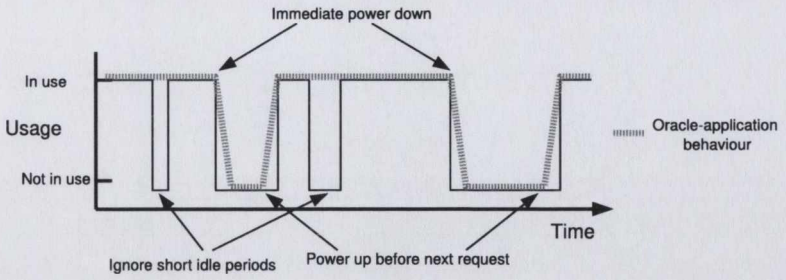

Figure 5.11: Oracle-application behaviour

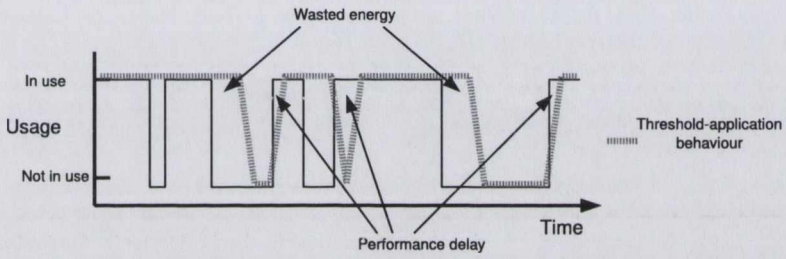

Figure 5.12: Threshold-application behaviour

just in time to service the next request from the user (Fig. 5.11). The performance of this optimal application is a useful baseline to which to compare the performance of realisable applications.

At the opposite end of the energy-efficiency spectrum is the *always-on* application (Harris, 2007). This application makes no attempt to conserve energy by suspending during idle periods. The difference in energy consumption between the Oracle and always-on applications is the maximum that can be saved through power management. We use the energy consumption of the always-on application as an upper bound when comparing applications.

5.6.6.2 Threshold applications

Threshold applications are the standard for power management of devices (Harris, 2007). These applications simply wait until an idle period exceeds some predefined threshold before powering down the device (Fig. 5.12) and energy is wasted while waiting for the threshold to pass. These applications do not attempt to predict when the device will be used next so the user must manually reactivate the device.

The original study implemented applications with thresholds of 5, 10, 15, 20, 25 and 30 minutes. In addition to these applications the original study experimented with a variation on standard threshold applications. The Standby/Wakeup on Bluetooth (*SWOB*) application adds a threshold on user

Application	Idle time (IT)	Bluetooth (BT)	Object range (OR)	Face detection (FD)	Voice analysis (VA)
IT	✓				
IT, BT	✓	✓			
IT, OR	✓		✓		
IT, BT, OR	✓	✓	✓		
IT, BT, FD	✓	✓		✓	
IT, BT, OR, FD	✓	✓	✓	✓	
IT, BT, OR, FD, VA	✓	✓	✓	✓	✓

Table 5.5: Fixed-sensor sets

presence to the standard threshold on idle time (Harris & Cahill, 2005b). This application uses the bluetooth sensor to detect the user's presence. When the idle time exceeds 1 minute and the user has not been detected in the last 5 periods the monitor is suspended. The monitor is activated when the user is detected again. We evaluate the same basic threshold applications as the original study (*T05*, *T10*, *T15*, *T20*, *T25*, T30) and also the *SWOB* application for comparison.

5.6.6.3 Fixed-sensor applications

The original study compared the performance of seven fixed-sensor applications to threshold applications using Bayesian networks to learn contexts Harris (2007). Table 5.5 shows the set of sensors used in each application. In our experiments KAFCA is used to learn accurate context definitions for a fixed set of sensors. The first phase of KAFCA is carried out as normal, and defines context definitions each for individual sensor. The second phase of KAFCA is restricted so that only one combination of sensors is considered for each application. The policy that is output from KAFCA defines how an application should select actions given sensor data from its particular combination of sensors.

5.6.6.4 Run-time sensor selection application

In addition to fixed-sensor applications the experiment also evaluates a *sensor-selection* application where sensors are selected at run-time using KAFCA. In the case of this application the second phase of KAFCA is carried out as described in Section 3.6, and the policy that is output defines how the application should select actions given sensor data from the selected, most suitable set of sensors.

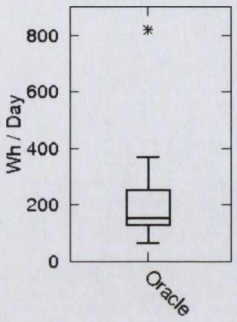

Figure 5.13: Total energy consumption

5.6.6.5 Evaluation metrics

The performance of each application (oracle, always on, threshold, fixed sensor, and sensor selection) is evaluated against a number of application-specific metrics. Each application runs for the duration of the recorded sensor data and the results of its performance are gathered. We apply similar metrics to the original study to evaluate energy efficiency, user-perceived performance, and the impact on device lifetime.

The *delta-energy-consumption-from-Oracle* metric measures the difference between the energy consumption of an application and the energy consumption of the Oracle application. This metric measures the energy wasted by an application compared to the Oracle application, which wastes no energy. User-perceived performance is measured using two metrics. If the application incorrectly suspends when the monitor is *in use* then the user experiences a *false suspend* (FS) and must wait for the monitor to reactivate. If the device does not automatically activate at the end of an idle period then the user experiences a performance delay and must perform a *manual activate* (MA). Occurrences of these events are counted to measure user-perceived performance. The impact of an application on device lifetime is measured by counting the number of times the device suspends but the time suspended is less than the breakeven period. This is referred to as the *breakeven-not-reached* (BNR) metric.

5.6.7 Results for recorded sensor data

In these experiments bar-and-whisker plots are again used to summarise the results from eighteen users for which there is recorded data. As in the sentient-couch scenario the median is used as the mean is sensitive to outliers. All experimental results have been averaged so they represent application

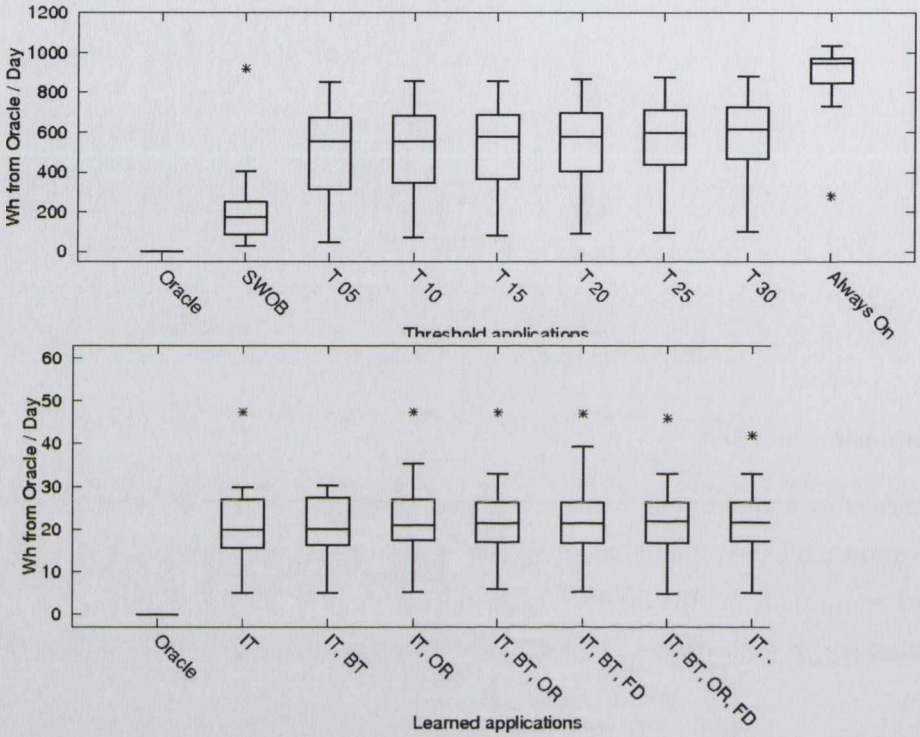

Figure 5.14: Delta energy consumption from the Oracle application

performance per day.

Fig. 5.13 summarises the energy consumption per day of users in the study. The median consumption was 152Wh, which equates to approximately 3 hours usage per day. Most users consume between 64 and 368Wh on their computer monitors, i.e., between 1 and 7 hours per day approximately. A single outlier uses 818Wh per day, which equates to ~17 hours of usage per day. It would seem that there was some error when recording the data for this user as it is very unlikely that a user would work an average of 17 hours per day over five consecutive days. Regardless of this outlier the range in energy consumption of other users shows that individual users have very different device-usage patterns.

5.6.7.1 Energy savings

Fig. 5.14 summarises the energy efficiency of each application relative to the Oracle application. The results for threshold and learned applications are displayed in separate graphs due to space con-

straints. With the exception of the *SWOB* application the threshold applications perform similarly in terms of energy savings. Their median values are in the range 551–618Wh from the Oracle application. The *SWOB* application performs significantly better with a median value of 173Wh from the Oracle. The *SWOB* application suspends the monitor after just one minute of an idle period so this difference in performance suggests there are many idle periods between 1 and 5 minutes long which the monitor can suspend for.

The learned applications perform very well in comparison to the threshold applications (note the change in scale on the y-axis). Their median values range between 19.9 and 21.8Wh from the Oracle. Such a small variation in performance suggests that the sensors used by each application had very little effect on performance under this metric. The application that used only the idle-time sensor (*IT*) performed better than the application that used all sensors (*IT, BT, OR, FD, VA*). The idle-time sensor detects idle periods with 100% accuracy and is therefore very suitable for selecting when to *suspend* the monitor and save energy. This suitability was confirmed when we examined the sensors selected for users by the *sensor-selection* application. In every case the idle-time sensor was selected as part of the most suitable set. The *sensor-selection* application's median value is 21.3, which is a 7% worse performance than the best fixed-sensor application (*IT*, median = 19.9).

5.6.7.2 User-perceived performance

User-perceived performance is measured using the FS and MA metrics. The results of the FS metric are shown in Fig. 5.15. It is immediately obvious that none of the threshold applications cause any FSs. This is to be expected as their encoded logic explicitly specifies that all suspends occur when the device is idle. In contrast almost all of the learned applications cause FSs for some users. Only the learned application where just the idle time sensor is used (*IT*) shows no FSs. This is unsurprising as this application is only influenced by the idle time, much like a standard threshold application. In other learned applications the idle time is not the only influence on action selection and some FSs occur. This is particularly obvious in the case of the application that uses all sensors (*IT, BT, OR, FD, VA*). However it should be noted that all learned applications show a median of 0 FSs per day, and that many users do not experience FSs at all. The *sensor-selection* application performs quite well, and it is noteworthy that its overall range of results (0–0.81) is significantly smaller than the ranges of other learned applications excluding *IT* (0–4.1 is the next smallest). This suggests that by selecting the most suitable sensors for each user the number of FSs they experienced was reduced although the result is not conclusive.

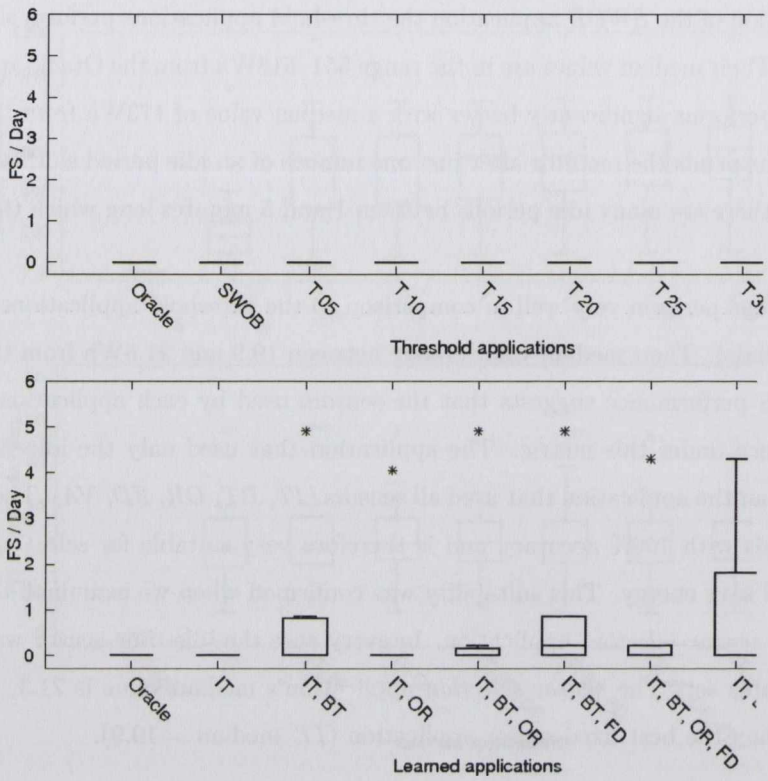

Figure 5.15: False suspends

The results of the MA metric are shown in Fig. 5.16. Threshold applications that use only idle time do not attempt to activate the monitor before the user requires it, therefore every idle period in which they suspend causes an MA. The longer the thresholds of these applications the fewer times they suspend, and therefore the fewer MAs they cause. This trend can be seen across the applications from *T05* to *T30*, as the median number of MAs falls from 3.9 to 1.9 per day. The *SWOB* application performs very well under this metric and only causes a median of 0.4 MAs per day.

In comparison to the threshold applications the learned applications perform very badly under this metric. The *IT* application has the highest median number of MAs at 11.01 per day. As observed previously this application behaves much like a threshold application as it is only influenced by the idle time. Therefore it initially seems strange that this application causes so many MAs compared to threshold applications. On closer examination of the behaviour of the *IT* application this was explained by more frequently suspending than standard threshold applications. More frequent suspensions of the monitor meant more MAs when the user returned. This also accounted for the

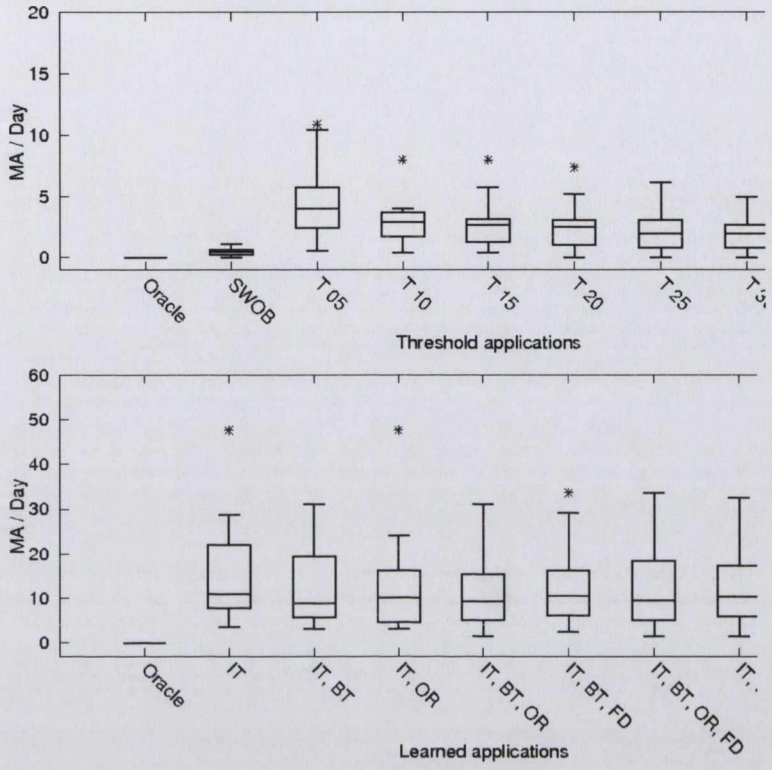

Figure 5.16: Manual activations

increased numbers of MAs in other learned policies, as they also suspend far more frequently than the threshold policies.

Interestingly, the *IT* application was not greatly outperformed by the other fixed-sensor applications or the *sensor-selection* application, even though it did not attempt to activate at the end of idle periods. The median values of learned applications are between 8.9 and 10.6 MAs per day. This suggests that the other sensors did not help the application to activate before the user returned to the monitor. The *sensor-selection* application's median value of 10.3 was 16% worse than that of the best fixed-sensor application (*IT, BT* median = 8.9).

5.6.7.3 Impact on device lifetime

The impact on device lifetime is measured using the breakeven-not-reached (BNR) metric and the results of this metric are shown in Fig. 5.17. The threshold applications again perform well under this metric. Even the worst performing threshold application (*T05*) has a median of just 0.4 BNRs

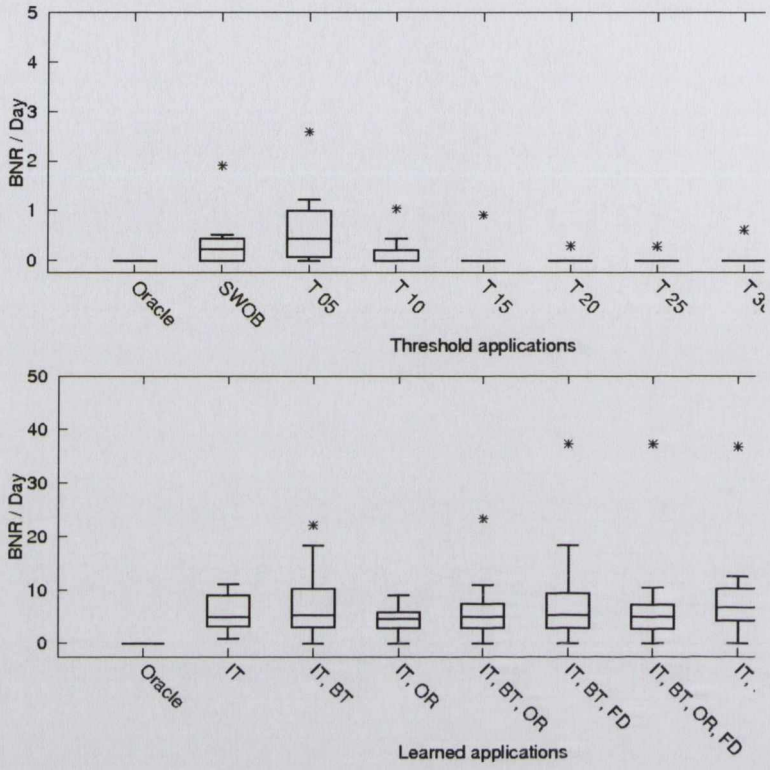

Figure 5.17: Breakeven not reached

per day.

In comparison the learned applications again perform badly under this metric. The reason for these results is the same as for the MA metric, i.e., these applications cause more frequent suspends and are therefore more likely to incur BNRs. All of the learned applications perform similarly, with median values between 4.7 and 6.3 BNRs per day. This suggests that the sensors used by different applications did not help the application to avoid suspending when it would result in a BNR. The *sensor-selection* application's median value (4.6) is the same as the best fixed-sensor application (*IT, OR*).

5.6.7.4 Conclusion

These results reveal a number of interesting points. Firstly they show that there is a trade off between energy conservation and user performance/device lifetime. The learned applications significantly outperform the threshold applications in terms of energy efficiency, but are outperformed by the threshold applications based on user-perceived performance and impact on device lifetime. The learned appli-

cations cause an increased number of MAs and BNRs as they suspend the device more frequently to reduce its energy consumption.

The *sensor-selection* application shows very little difference in performance compared to fixed-sensor applications. Significantly the *IT* application which uses only the idle-time sensor performs almost as well as other learned applications that use a variety of sensors. This suggests that the other sensors do not assist the application in identifying the context, in particular to avoid MAs and BNRs. To better understand this we must analyse the usefulness of sensor data to the application.

5.6.8 Sensor-data analysis

Our analysis of the recorded sensor data is intended to identify which sensors provide data that is suitable for identifying contexts. We start by measuring the correlation between sensor data and idle time, and then analyse the patterns of sensor data that are observed in different contexts.

5.6.8.1 Correlation

Correlation is a commonly-used approach to evaluate the relationship between data. It is often understood to mean any relationship between variables, but more formally statistical correlation (usually measured as a correlation coefficient) indicates the strength and direction of a linear relationship between two random variables (Baron, 2007). It measures the degree to which one linear variable can predict the value of another. There are a variety of methods for calculating correlation coefficients. We apply the well-known Pearson product-moment correlation coefficient (Moore, 1999). The coefficient output ranges from -1 to +1. A coefficient of +1 implies that a linear equation describes the relationship between the first and second variables perfectly– every increase in the value of the first variable is mirrored by an increase in the value of the second variable. A coefficient of -1 also implies perfect correlation, where every increase in the value of the first variable is mirrored by a decrease in the value of the second variable. A coefficient of 0 implies there is no linear relationship between the variables.

In this scenario it is obviously important that sensor data can be used to predict when the monitor will be idle, i.e., that there is a correlation between sensor data and the idle time. Only two of the sensors used in this scenario fulfill the linear requirement– the bluetooth and face-detection sensors. Intuitively the values from these sensors should increase as idle-time increases. Fig. 5.18 shows the measured correlation between the bluetooth and face-detection sensors with idle time. The figure summarises the coefficients as measured across *all* periods and also across only *idle* periods. It is impossible to measure the correlation across only active periods, as the idle value is always zero in

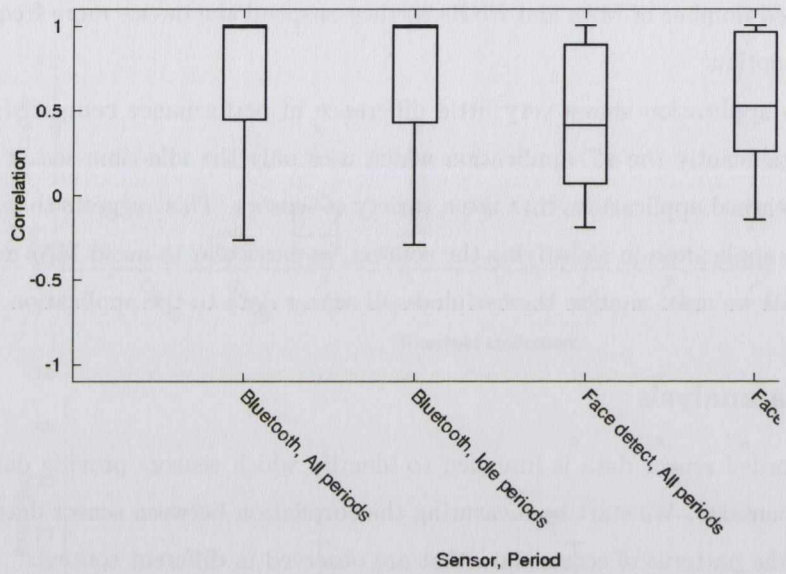

Figure 5.18: Correlation between bluetooth, face-detection sensors and idle time

these periods and this causes division-by-zero issues during coefficient calculations.

Each bar-and-whisker plot summarises the coefficients calculated for individual users. The first thing of note is the wide variation in correlation across users. In all plots the correlation for different users may be complete ($\sim$ 1) or nonexistent ($\sim$ 0). This demonstrates that in this scenario sensor suitability is dependent on the user and/or environment. The bluetooth plots for *all* and *idle* periods are very similar (median 0.99), which suggests that the bluetooth sensor is equally effective for detecting the user when active and idle. The face-detection sensor has a lower correlation for both *all* and *idle* periods (medians 0.41, 0.53) suggesting that it is less reliable for detecting the user. The difference between the *all* and *idle* period plots suggests that it is better at detecting when the user is away than when they are present.

Statistical correlation is a commonly used metric for evaluating the relationship between variables however its usefulness for this scenario is limited as it can only be applied to two of the four available sensors. It also provides a general measure of how correlated sensor data is but does not capture how useful the sensor data is, e.g., it does not capture how consistently sensor data predicts the user's return in the *about-to-use* period. In order to analyse the sensor data further we examine patterns that occur in the data during specific periods.

	User detected	User undetected
Bluetooth	0	>0
Face detection	0	>0
Object range	30-80	90-160
Voice activity	0-40	0-40

Table 5.6: Patterns of sensor data from (Harris, 2007)

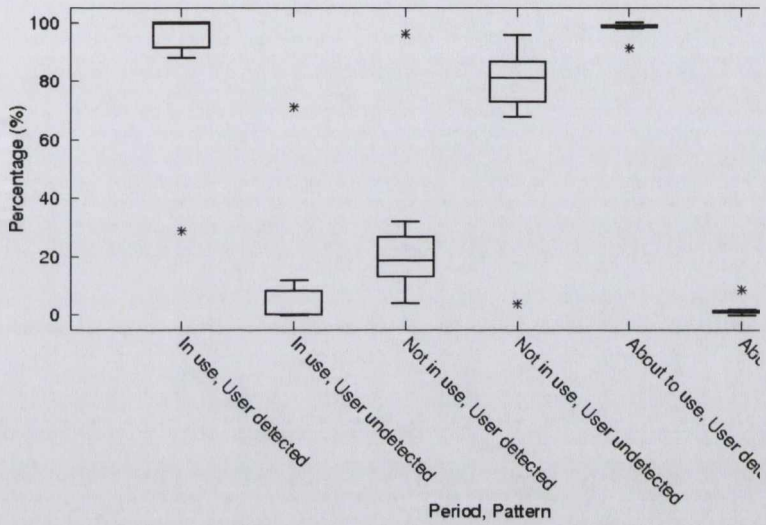

Figure 5.19: Bluetooth pattern occurrences

5.6.8.2 Pattern occurrences

In this section we examine the patterns of sensor data that occur in different periods within the recorded data, and their implications for a power-management application. In the original study the author identified patterns in sensor data that indicated when the user was detected at the device. These patterns are summarised in Table 5.6. The original study observed that the voice-activity sensor consistently output readings in the range 0-40, regardless of whether the user was present or not. This lead them to conclude that this sensor was useless for detecting the user.

The occurrences of these patterns are analysed for three periods– *in use* , *not in use* and *about to use*. These periods were previously described in our discussion of the reward model (Section 5.6.4). We select these periods as in each a different application action is appropriate. During *in-use* periods the application should *defer* suspending, during *not-in-use* periods the application should *suspend*, and during *about-to-use* periods the application should *activate*. We analyse the occurrences of patterns in these periods for each sensor.

Pattern occurrences for the bluetooth sensor are shown in Fig. 5.19. Each plot summarises the

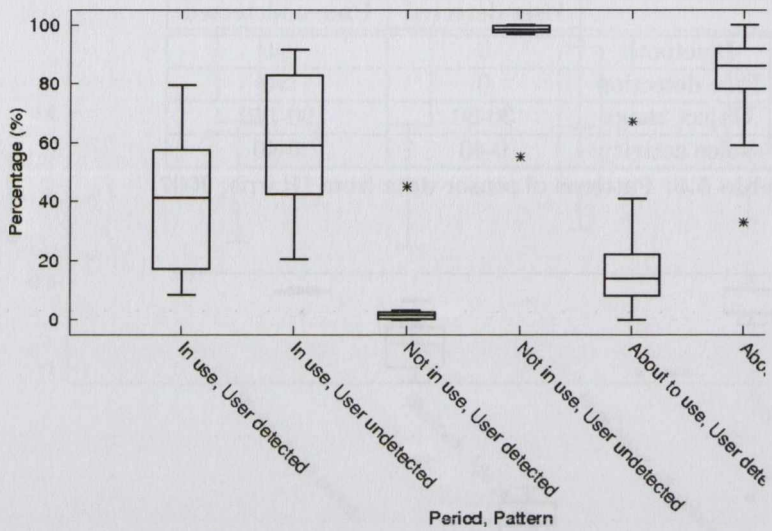

Figure 5.20: Face-detection pattern occurrences

proportion of user-detected to user-undetected patterns in each of the three periods as a percentage. As expected a high percentage of the patterns in *in-use* periods indicate that the user is detected (median 99%). Similarly a high proportion of the patterns in *about-to-use* periods indicate the user is detected (median 99%). This is significant as it means the bluetooth sensor can detect the user before they require the monitor. However the *not-in-use* period results show that a significant proportion of patterns indicate the user is detected despite the monitor being inactive (median 19%). These occurrences may be a result of the user leaving the bluetooth dongle at the computer while they are away, or working in the proximity of but not at the monitor.

An equivalent set of plots for the face-detection sensor are shown in Fig. 5.20. It is immediately obvious that this sensor is significantly worse at detecting the user in *in-use* periods than the bluetooth sensor. A low percentage of the patterns in *in-use* periods indicate that the user is detected (median 41%). The wide range of percentages (8–80%) suggests that this sensor is very dependent on the particular user and/or environment. An incorrectly-oriented camera or a room with low-light conditions could be responsible for poor face-detection rates (Harris, 2007). In contrast this sensor is very effective for detecting that the user is not present during *not-in-use* periods (median 99%). It rarely detects the user in *about-to-use* periods (median 14%). This may be a consequence of the processing delay when identifying a face in an image, described in the original study.

The pattern-occurrence plots for the object-range sensor are shown in Fig. 5.21. The sensor is reasonably good at detecting the user in *in-use* periods (median 86%), but the range of detection rates

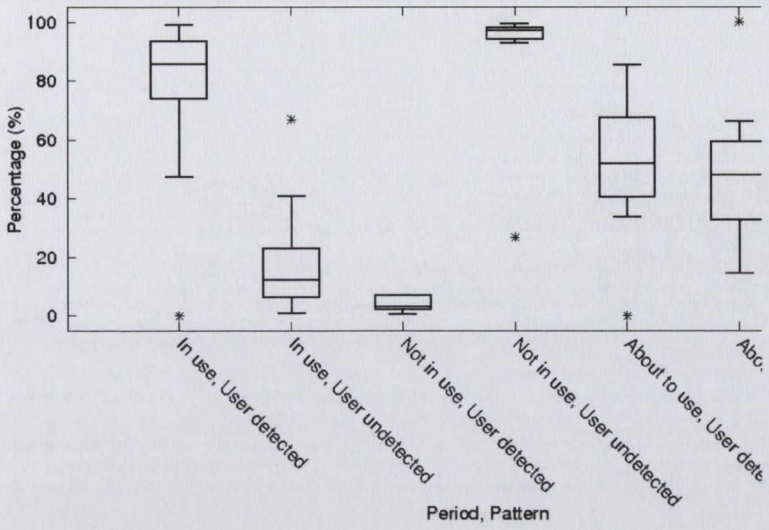

Figure 5.21: Object-range pattern occurrences

is wide (47–99%). Again this variation may be explained by incorrectly-oriented sensors. In (Harris, 2007) the author notes that it was difficult to orient this type of sensor so that it detected the user consistently. Similar to the face-detection sensor this sensor is very effective at detecting when the user is not present (median 97%) in *not-in-use* periods, but is not very effective for detecting the user in *about-to-use* periods (median 52%).

As mentioned previously the original study was unable to identify voice-activity patterns that indicated the user was detected or undetected. For all periods the observed patterns were in the same range (0–40). Despite experimenting with a variety of possible ranges we were also unable to discover any patterns that indicate a user's presence or absence. As a result we must concur with the original study's suggestion that the voice-activity sensor was not suitable for this scenario.

5.6.8.3 Conclusion

This analysis reveals some interesting details about the sensor data. The statistical-correlation analysis shows that the bluetooth sensor is in general very well correlated with the idle time and that the face-detection sensor is in general much less correlated. However correlation has limited usefulness for this scenario as it cannot be applied to all sensors and only provides a high-level measure of the relationship between sensor data and idle time (user presence).

Our analysis of the patterns of sensor data also show that the bluetooth sensor is very good at detecting the user when they are at the monitor, and that it also detects the user in the *about-to-use*

	Good user/environmental characteristic	Bad user/environmental characteristic
Idle time	Always reliable	–
Bluetooth	Carries dongle	Leaves dongle at desk
Face detection	Bright office	Dark office
Object range	Correctly oriented	Incorrectly oriented
Voice activity	Own office	Shared office

Table 5.7: User and environmental characteristics

period. However the user is frequently detected at other points *during* idle periods. This means that with this sensor an application will encounter a significant number of false positives where the bluetooth sensor indicates that the user is about to use the monitor. The face-detection and object-range sensors are less effective for detecting the user's presence, and their suitability varies widely across different users (and environments). Both consistently detect when the user is not present, but the face-detection sensor is poor at detecting the user in the *about-to-use* period and the object-range sensor is mediocre at best. Data from the voice-activity sensor has no identifiable pattern.

Based on this analysis we conclude that the recorded sensor data provides very limited opportunities to test the sensor-selection phase of KAFCA, as most sensors do not consistently provide suitable sensor data for identifying the context. To test this phase of KAFCA more completely we decided to generate sensor data for theoretical users that had different personal and environmental characteristics.

5.6.9 Sensor-data generation

To further evaluate KAFCA sensor data was generated for six different users. Each user had a unique combination of user and environmental characteristics that dictated the sensors that were suitable to them, e.g., they might be a conscientious user who always carries the bluetooth dongle, or a user who works in a dark office and whose face cannot be detected. The possible characteristics are shown in Table 5.7. Each sensor, with the exception of the idle-time sensor, may provide *suitable* or *unsuitable* sensor data for identifying the context. If a user has the *good* characteristic associated with a sensor then their generated sensor data for that sensor will be a good indicator of their context, and vice versa for the *bad* characteristic.

Table 5.8 shows the six users and the sensors for which they have *good* characteristics. The idle-time sensor is suitable for all users as it is completely reliable for detecting the idle period, independent of the user or environment. Sensor data is generated for each user according to these characteristics.

The sensor-data generation process generates sensor data for five days for each user, the same

	Idle time	Bluetooth	Face detection	Object range	Voice activity
user1	✓				
user2	✓	✓	✓	✓	✓
user3	✓	✓			
user4	✓		✓		
user5	✓			✓	
user6	✓				✓

✓= *good* characteristic for this sensor

Table 5.8: Characteristics of generated users

amount as the recorded sensor data from the original study. The first step is to generate the idle periods for a user. This process is similar to that used to simulate the current context in the sentient-couch scenario, i.e., it uses Monte Carlo simulation based on a discrete-probability distribution (Listing 5.6). The probability of no idle period is 0.99, the probability of a short idle period is 0.005, the probability of a medium idle period is 0.04, and the probability of a long idle period is 0.001. Starting at the beginning of generated time the *nextIdlePeriod()* method is called every five seconds until an idle period is generated. Generated time moves forward to the end of the generated idle period and the process starts again. Once the end of the 5 days of generated time is reached this generation process stops.

```
1  //data generation {no idle period, short period, medium period, long period}
2  //select two random number between 0 and 1
3  double random = (double)rand() / RAND_MAX;
4  double random2 = (double)rand() / RAND_MAX;
5  double idlePeriodLength;
6
7  //p(no idle period) = 0.99
8  if (random<0.9){
9    idlePeriodLength = 0;
10 }
11 //p(short period) = 0.005
12 else if (random < 0.995){
13   //idle period between 30 and 60 seconds long
14   idlePeriodLength = 30 + floor(30*random2);
15 }
16 //p(medium period) = 0.004
```

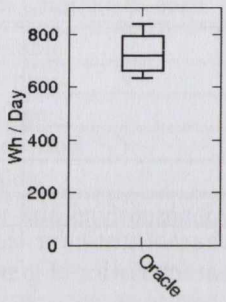

Figure 5.22: Total energy consumption

```
17 else if (random < 0.999){
18   //idle period between 1 and 10 minutes long
19   idlePeriodLength = 60 + floor(9*60*random2);
20 }
21 else {
22   //idle period between 10 and 60 minutes long
23   idlePeriodLength = 10*60 + floor(50*60*random2);
24 }
```

Listing 5.6: The `nextIdlePeriod()` method

Once the idle periods are generated the rest of the sensor data can be generated accordingly. A new value is generated for each sensor every five seconds and written to the user's log files. The value generated for a sensor depends on the current idle period, the user's characteristic for that sensor, and the sensor's error rate. For example the bluetooth sensor in an *in-use* period, for a user with the *good* characteristic for this sensor (always carries the bluetooth dongle), will generated the value 0 with an error rate of 14%. The values for sensor error rates are taken from the original study (Harris, 2007).

Although the original study observed that the voice-activity sensor did not produce any useful data we define patterns for this sensor so that its generated sensor data is suitable for some users. For users that have a *good* characteristic for this sensor it produces data in the range 0–10 when the user is away from the monitor and 10–40 when the user is at the monitor. The motivation for this is simply to create additional scope for sensor selection in the scenario.

5.6.10 Results for generated sensor data

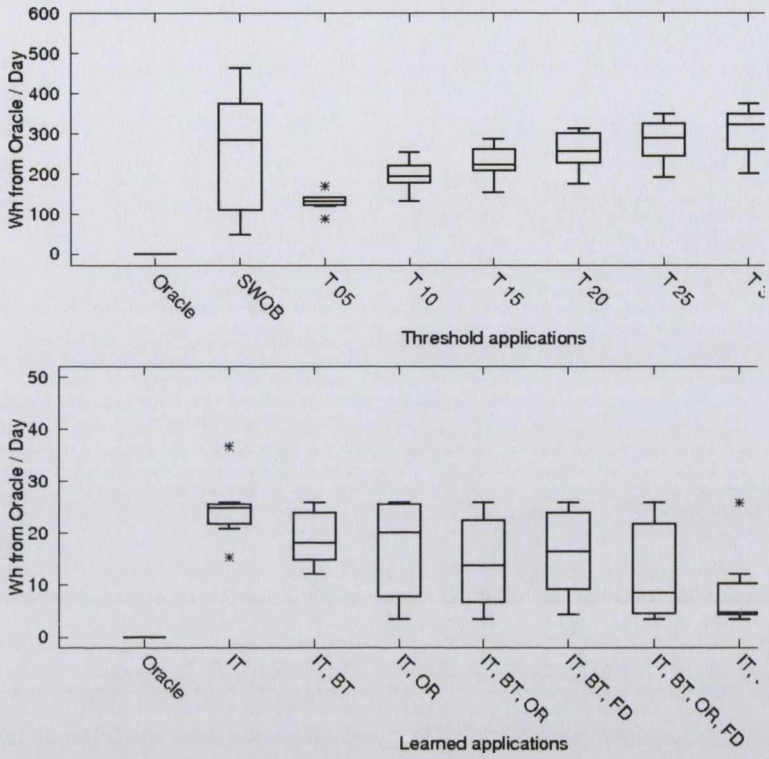

Figure 5.23: Delta energy consumption from the Oracle

We apply the same approach to experiments using generated sensor data as was applied to the recorded sensor data. Again bar-and-whisker plots are used to summarise the results, and all results have been averaged so they represent application performance per day. Fig. 5.22 summarises the energy consumption per day of generated users. They consume between 636 and 839 Wh per day, which demonstrates that individual generated users have different device-usage patterns.

5.6.10.1 Energy savings

Fig. 5.23 summarises the energy efficiency of each application relative to the Oracle application. As we observed with the recorded sensor data the performance of standard threshold applications degrades as the threshold time increases from 5 minutes to 30 minutes. In contrast to the experiments with real sensor data the *SWOB* application performs worse than standard threshold applications for the generated data. Our analysis of the recorded sensor data showed that the bluetooth sensor was suitable for most users. In contrast the generated bluetooth sensor data is only suitable for two out

of the six generated users. The other four users have the "leave the bluetooth dongle at their desks" characteristic, therefore their presence is incorrectly detected and the monitor is not suspended by the *SWOB* application.

As was the case for recorded sensor data, the learned applications perform very well in comparison to the threshold applications. In the experiments with recorded sensor data there was very little variation in learned-application performance as all applications based their decisions to *suspend* the monitor on the idle-time sensor. In these experiments we observe that other sensors contribute to this decision and therefore there is more variation in performance. The *IT* application waits for some idle time to pass before suspending to avoid incurring BNRs in short idle periods. In comparison other sensors can detect that the user has left the device and applications that use those sensors can suspend earlier in the idle period.

The general trend in Fig. 5.23 shows that the more sensors are used in an application the better its performance. The application with one sensor (median = 24.9) performs worse than applications with two sensors (medians = 18.0, 20.1), which perform worse than applications with three sensors (medians = 13.7, 16.5) and so on. It makes sense that applications with more sensors perform better as they are more likely to use a sensor that matches the good characteristics of different users.

The best performing applications are the *IT, BT, OR, FD, VA* application and the *sensor-selection* application, with medians of 4.7 and 4.8 respectively. The outlier for both applications is *user1* – the user which has no good sensor characteristics. The *IT, BT, OR, FD, VA* policy has a sensor that suits the characteristics of almost every generated user therefore it is no surprise that this application performs very well. The *sensor-selection* application performs similarly well, wasting just 2.1% more energy than the *IT, BT, OR, FD, VA* application. The *IT, BT, OR, FD, VA* application is an unfair comparator as it uses all available sensors, therefore it is impossible for the *sensor-selection* application to select more suitable sensors for users. In fact the *sensor-selection* application selects just two sensors for most users, the idle-time sensor plus a sensor that matches their good characteristics (Table 5.8). If we compare the performance of the *sensor-selection* application to fixed-sensor applications that use just two sensors (*IT, BT* and *IT, OR*) there is a significant difference in performance. The median energy wasted by the *sensor-selection* application is 73% and 76% less than these fixed-sensor applications.

5.6.10.2 User-perceived performance

The results of the FS metric for generated sensor data are shown in Fig. 5.24. Neither threshold nor

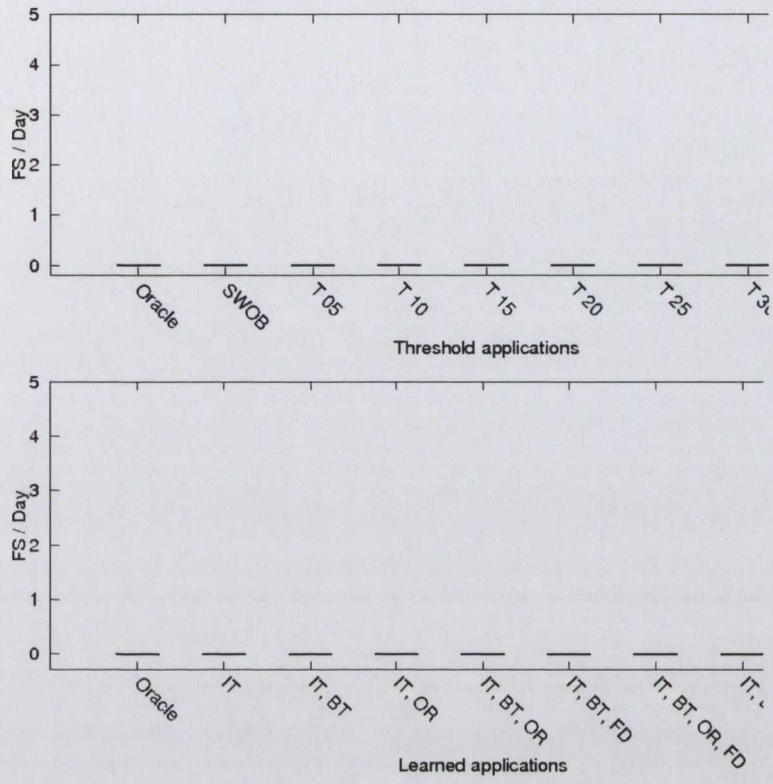

Figure 5.24: False suspends

learned applications cause any FSs. This was also the case for threshold applications using recorded sensor data, while learned applications that used recorded sensor data experienced a small number of FSs. Our analysis of sensor data showed that sensors were not always consistent at detecting the user's presence while the monitor was *in use*. The reduction in FSs for generated sensor data shows that the generated sensor data more consistently indicates a user's presence.

The results of the MA metric are shown in Fig. 5.25. As described previously the standard threshold applications do not attempt to activate the monitor therefore each of their suspends causes an MA. The number of MAs decreases as the threshold increases as fewer suspends occur. The *SWOB* application again performs well under this metric, although as already observed this application only suspends the monitor for users that have the good bluetooth characteristic, therefore it also only activates the monitor for these users. It causes no MAs for the other users.

In comparison to the threshold applications most learned applications perform badly under this

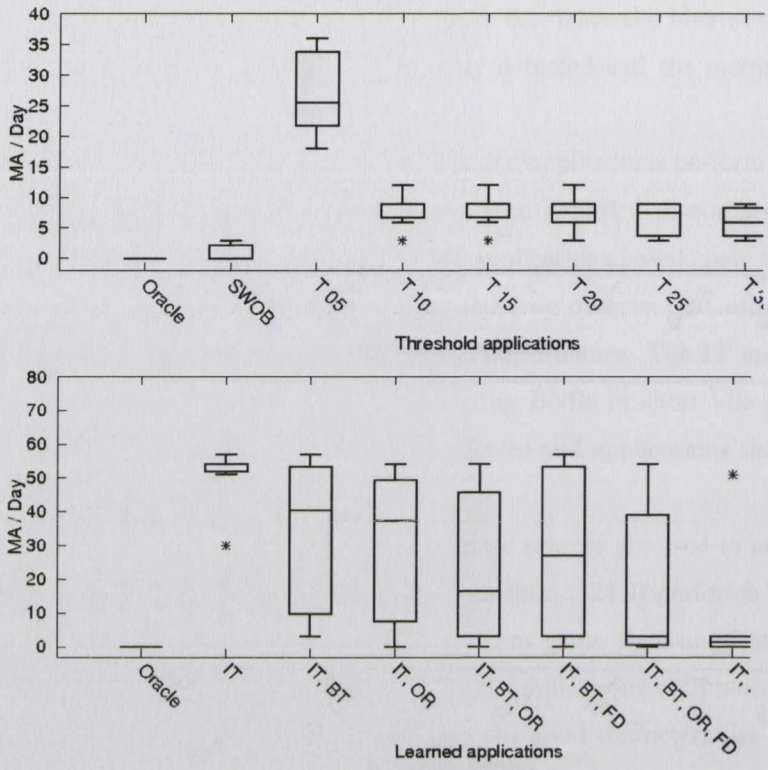

Figure 5.25: Manual activations

metric. The *IT* application again has the highest median number of MAs at 54 per day. As stated previously this application behaves like a threshold application as it is only influenced by the idle time therefore every suspend causes an MA. The other learned applications perform much better as their sensors detect the user in the *about-to-use* period. As observed in the discussion of energy savings (Section 5.6.10.1) application performance improves as the number of sensors they use increases. Again the *IT, BT, OR, FD, VA* and *sensor-selection* applications perform best with medians of 1.5 and 0 MAs per day respectively. The outlier for both applications is *user1*, the generated user that has no good characteristics for sensors. The *sensor-selection* application performs better than the *IT, BT, OR, FD, VA* application under this metric, however this cannot be explained by sensor unsuitability and must be a function of some inaccurate policies being learned. If we compare the *sensor-selection* application to other fixed-sensor applications that use two sensors there is again a significant difference in performance. The *IT, BT* and *IT, OR* applications cause 40.5 and 37.5 more MAs per day respectively compared to the *sensor-selection* application.

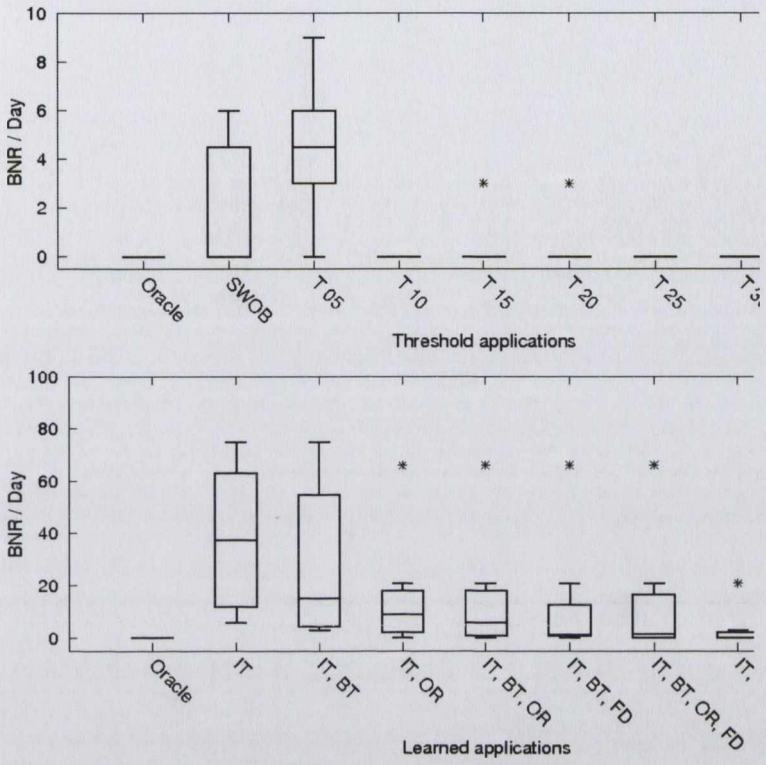

Figure 5.26: Breakeven not reached

5.6.10.3 Impact on device lifetime

The results of the BNR metric are shown in Fig. 5.26. Again most threshold applications perform well under this metric. The relatively poor performance of the *SWOB* and *T05* applications is explained by the existence of more short idle periods in the generated data than in the recorded data.

In comparison to the threshold applications most of the learned applications perform badly under this metric. As discussed earlier this is explained by learned applications suspending the monitor more frequently. The worst performing is again the *IT* application, which has no sensor data that informs it of the user's proximity to the device. All of the other learned applications outperform this application as they have some sensor that detects the user's presence. Again the performance of applications improve as more sensors are used. The *IT, BT, OR, FD, VA* and *run-time-selection* applications perform best, both with median 0 BNRs per day.

5.6.10.4 Conclusion

The second objective of the thesis is to select suitable sensors for identifying application contexts at run time and this objective is achieved by applying the second phase of KAFCA. These results show that KAFCA can successfully select suitable sensors at run time. Where users and environments have varying characteristics, and sensors have varying suitability depending on these characteristics, that application performance can be improved by selecting the most suitable sensors at run time. Application contexts are identified more accurately by selecting sensors than by using predefined, fixed sets of application sensors. This indicates the effect of achieving the second thesis objective.

Our analysis of the recorded sensor data showed that it did not have sufficient variation in its suitability to show the effect of sensor selection. The generated sensor data for users is based on explicitly defined user characteristics that define the suitable sensors for a user, therefore there is significant variation in sensor suitability and the effects of sensor selection are obvious in the results.

The *sensor-selection* application selected sensors that matched the underlying characteristics of each user. For each metric the performance of the *sensor-selection* application outperformed all learned applications except *IT, BT, OR, FD, VA*. This application uses all available sensors so it is impossible to select better sensors, however it is unlikely that deployed applications will be designed to use all available sensors, given the uncertainty in the run-time environment (Section 1.3.2).

5.7 Generalisability of results

The results observed in these scenarios exhibit both the pros and cons of our approach. The evaluation is application specific in each experiment, however we draw some general indications from the observed results. Reinforcement learning appears to be a viable approach for learning interpretations of sensor data, however the accuracy of interpretations is heavily dependent on the reward model and parameterisation of the learning algorithm, as well as on the consistency of the underlying sensor data. Discrete-state refinement has the potential to significantly improve the accuracy of learned context definitions but the overhead of learning policies for very refined discrete states may become prohibitive. Sensor selection is an effective technique where users and/or the environment exhibit significantly varied characteristics, and where sensors contribute significantly varied information about the context.

5.8 Performance and scalability

The results of our experiments in Section 5.4.5 show that there is a significant overhead involved in learning new policies. This is a issue that is common among reinforcement-learning algorithms Sutton & Barto (1998), as they learn from an initial position of having zero knowledge about how to behave. In this thesis reinforcement learning was selected on the basis that it is the technique that is least dependent on expert knowledge and therefore has the potential to provide greater context-aware application autonomy at run time. It facilitates our approach to interpret sources of sensor data that were unforseen by the developer as its reward model can be used to learn policies for any underlying discrete states and sensor space. However our learning process is particularly slow due to the learning rate we apply to overcome spatial and temporal dependencies. This slows learning significantly as a very small proportion (0.01) of each reward gathered is considered when updating Q-values, and this exacerbates the performance issue associated with reinforcement learning.

Another concern that is common with learning approaches is their scalability. As the number of inputs increases the learning task can grow to infeasible sizes Callan (2003). In our approach we apply apply two levels of generalisation to address this issue. In phase one of KAFCA we generalise about sensor data using discrete states, and in phase two we further generalise about discrete states using contexts. The effect of this is to reduce the number of states that the reinforcement learning algorithm must consider when learning a policy.

5.9 Chapter summary

This chapter described the evaluation of KAFCA, our approach to knowledge autonomy for context-aware applications. The evaluation process in this chapter had three objectives: to evaluate reinforcement learning as a technique for learning accurate policies for discretised sensor, to evaluate how context-definition accuracy was improved by applying KAFCA to refine discrete states, and to evaluate the improvement in application performance where a context-aware application selected its sensors at run-time using KAFCA instead of depending on a predefined set of sensors.

The *line* and *grid* scenarios evaluate reinforcement learning as an approach for learning about sensor data. These simple scenarios evaluated if accurate policies could be learned for discretised sensor data. The *line* scenario revealed that inconsistency within discrete states could cause a *spatial dependency* that introduced inaccuracy into learned policies. The *grid* scenario revealed that the sequence in which discrete states were visited could cause a *temporal dependency* that also introduced inaccuracy into learned policies. Both of these issues were successfully addressed by using a very low

learning rate.

The sentient-couch scenario evaluated the accuracy with which context definitions could be learned, and the effect of discrete state refinement on accuracy. This scenario showed that discrete-state refinement improved the accuracy of context definitions by up to 95% and that most of this improvement (68%) occurred in the first few refinements. However it also showed that the learning overhead increased by ~47% per refinement. The first refinement provided 10 times the accuracy improvement per learning iteration as the fifth refinement did.

The power-management scenario evaluates the effect on application performance of selecting sensors at run-time compared to using a predefined set of sensors. Initial experiments were based on recorded sensor data however the lack of variation in the performance of applications that used different sets of sensors caused us to carry out an analysis of the recorded data. This revealed that most of the available sensors were not suitable for identifying the application's contexts. In order to evaluate sensor selection more completely sensor data was generated for a group of very different users. Each user had different characteristics that dictated the sensors that provided suitable data for identifying their context. This lead to more interesting results from applications that used different sets of sensors. For each metric we observed that application performance improved as the number of sensors used by an application increased. The performance of the *sensor-selection* application was only matched by that of the application that used all available sensors, even though the *sensor-selection* application mainly selected just two sensors for each user. In comparison to fixed-sensor applications that also used two sensors the *sensor-selection* application performed very well. It wasted 73% and 76% less energy than these applications, and also caused 0 MAs compared to their 40.5 and 37.5 per day.

Chapter 6

Conclusions

This thesis describes the design and implementation of an approach to knowledge-autonomous context-aware applications. Specifically, it describes a process that supports learning accurate context definitions and selecting the most suitable sensors at run time. This facilitates a context-aware application to adapt to its current run-time environment, identify its context more accurately, and as a result improve its performance. There is also a reduced dependency on the developer to encode expert knowledge in the application. This chapter summarises the achievements of the work and the contributions of the research to the state of the art. It concludes with a discussion of potential areas for future work.

6.1 Achievements

Our analysis of state-of-the-art approaches to context definition in Chapter 2 highlighted two limitations that motivated the work presented in this thesis. Firstly, the context definitions arising from these approaches depend to various degrees on encoded expert knowledge. These encoded structures limit the flexibility of context definitions and the ability to adjust context definitions at run time. Secondly, these approaches limit an application to predefined sensor types for identifying the context. This meant that issues such as sensor unavailability or unsuitability could not be addressed at run time. State-of-the-art approaches are dependent on and limited by expert knowledge encoded at design time.

This thesis presents an approach to knowledge-autonomous context awareness designed to address these limitations of state-of-the-art approaches. Chapter 3 described the design of our approach to learning context definitions at run time. Initially, we discussed existing definitions of context, their

limitations for describing context for knowledge-autonomous applications, and a novel definition of context from an application's perspective. We also described a representation of context definitions based on context edges that is flexible at run time. We outlined the KAFCA process that applies this theory of context. At the core of this process is reinforcement learning, which we identified as the learning technique least dependent on expert knowledge. Reinforcement learning is used to learn policies that define the meaning of sensor data in terms of action selection. Around this learning technique we defined two processes to define accurate context definitions and select useful sensors at run time. The accurate-context-definition process focuses on the inaccuracy introduced into context-definitions by discretisation. Sensor data has to be discretised to facilitate reinforcement learning, however during discretisation some information is lost which might prevent the application distinguishing changes of context. The accurate-context-definition process iteratively refines how sensor data is discretised so that the accuracy of context definitions is improved. The sensor-selection process then selects sensor combinations, combines their individual context definitions, and evaluates application performance to measure the usefulness of the sensor combination. This measure feeds back into the search for the most suitable sensor combination.

The evaluation of the approach was described in Chapter 5. Four scenarios were implemented with different evaluation objectives. The first two scenarios, *line* and *grid*, evaluated reinforcement learning as a technique for learning accurate policies that described the meaning of sensor data. These scenarios showed that reinforcement learning successfully learned accurate policies as long as a sufficiently low learning rate was used to overcome spatial or temporal dependencies. The third scenario, based on the sentient couch, evaluated the accuracy with which context definitions could be defined using the accurate-context-definition process of KAFCA, and the trade off between accuracy and learning time. Results showed that the accuracy of context definitions increased by up to 95% and that most of this improvement (68%) occurred in the first few refinements. It also showed that the learning overhead increased by ~47% per refinement. The first refinement provided 10 times the improvement in accuracy per learning iteration as the fifth refinement did. The final scenario, a power-management application, evaluated the effect of selecting sensors at run time on application performance. The performance of an application that applied the entire KAFCA process to define contexts and select the most useful sensors was compared to standard knowledge-intensive power-management applications, as well as applications that used KAFCA to define contexts but were limited to fixed sets of sensors. Initial experiments were based on recorded sensor data. The results showed that there was little variation in the performance of the learning applications, but analysis revealed that this was a result of the underlying sensor data, which in general did not provide useful information for identifying the context.

Sensor data was then generated for theoretical users with very different characteristics, and the results of experiments with this data showed much greater variation in performance. Only an application that used a fixed set of all available sensors matched the performance of the application that selected sensors at run time. On examination of the sensors selected for each user it was observed that their sensors matched their defined characteristics, which confirmed that the KAFCA process was selecting the most useful sensors for users.

The main contributions of this thesis are summarised as:

- An overview of state-of-the-art approaches to context definition with particular attention to their dependence on expert knowledge, their ability to adjust context definitions at run time, and their ability to adapt to sensor unavailability and unsuitability.
- A novel definition of context from a context-aware application's perspective and a flexible representation of context based on context edges, both of which are independent of expert knowledge.
- An approach to knowledge-autonomous context definition that uses reinforcement learning to learn accurate context definitions and select suitable sensors at run time, eliminating the need for the developer to provide any part of context definitions.
- An evaluation of this approach which shows that it is possible to significantly improve the accuracy of context definitions by adjusting how sensor data is discretised, and also shows that it is possible to select sensors to suit different run-time environments and users. Both of these results improve the accuracy with which application contexts are identified and thereby improve application performance.

6.2 Objective achievements

In Chapter 1 we defined two thesis objectives– to accurately identify application contexts from sensor data interpreted at run time, and to select suitable sensors for identifying application contexts at run time. In our review of state-of-the-art approaches in Chapter 2 we considered four characteristics of related approaches, which captured how well each approach addressed the thesis objectives. We now review our own approach in terms of these characteristics. .

6.2.1 Knowledge autonomy

In Chapters 2 and 3 we identified reinforcement learning as the learning technique that is least dependent on expert knowledge, and therefore the most suitable for autonomous, context-aware applications.

With the hindsight of our implementation of the scenarios in Chapter 5 we can review our opinion of reinforcement learning. The reward model, for example, is a knowledge-intensive structure, which effectively must capture how the application should behave. This is a significant task, and its complexity is most likely beyond that required to define the application contexts at design time using a knowledge-intensive approach. However, once the reward model is defined it creates the potential to learn policies for the discrete states of any set of sensors at run time.

Other knowledge-intensive requirements for reinforcement learning include parameters for the learning algorithm and for identifying when sufficient learning has occurred. Finally, there is a requirement for an application-performance metric for sensor selection, which is also a knowledge intensive structure. Overall our solution has quite significant requirements for expert knowledge, however all of this knowledge can be encoded at design time unlike other learning approaches described in Chaper 2, therefore we consider it to have a greater degree of knowledge autonomy than existing approaches.

6.2.2 Flexibility of context definitions

In Chapter 2 we identified that existing approaches to learning context definitions were limited by their dependency on a fixed discretisation layer. One of the contributions of this thesis is a discretisation layer that is tailored to the run-time environment. This discretisation layer can be refined to discretse sensor data at the highest precision offered by the underlying sensor therefore we consider it to be completely flexible.

6.2.3 Selection of suitable sensors

In the related work we identified only two approaches that considered the relevance of sensor data, and these approaches considered particular sensor data values rather than complete sensors. Our approach facilitates the selection and interpretation of any sensor that fulfills the ordered-data criteria at run time. Therefore our approach is successful by this measure.

6.2.4 Sensor unavailability at run time

The approaches reviewed in Chapter 2 treated sensor unavailability as a resource-discovery problem, where a missing sensor could be replaced by another sensor of the same type. Our approach adapts to use whatever sensors are available in the run-time environment, therefore it implicitly addresses this challenge. However it is not designed to adapt quickly to sensors coming and going in the environment due to the learning time needed to adapt. Therefore, we consider this challenge moderately addressed

by our approach.

6.3 Guidelines and heuristics

Context-aware applications in which this approach could be successfully applied need to meet a number of requirements. Suitable applications would be those that benefit significantly from customisation, e.g., where tailoring the application to an environment or user is critical. They would also be intended for deployment in unpredictable environments where customisation is likely to be needed. Suitable applications for this approach would also not be affected by the learning time required to customise the application, and would not be affected by the unpredictability of application behaviour during learning.

The application would need to be of a reactive nature, as the underlying reinforcement-learning technique causes applications to react to changes in their environment rather than plan for future changes. Due to the trial-and-error nature of reinforcement learning the application could not be safety-critical, or would need to be restricted such that the actions it can take are not safety critical. For example a traffic-signal controller is obviously safety critical as it could potentially cause car crashes, however safe actions can be defined, e.g., setting a traffic signal to green is only valid when other, conflicting lanes have a red signal. Such programmatic limitations could allow a learning application to operate safely.

During our own experimentation we observed that the main challenge of applying this approach was in learning accurate policies using reinforcement learning. This task was difficult in two ways. Firstly, it was difficult to capture appropriate rewards in the reward model, and over thousands of iterations we observed that small changes in the reward model could have significant effects on the learned policy. The more complex the reward model the wider the range of changes that could be made to it. Our conclusion was that the simplest reward model was often the most effective, and that it was better to start with a model that was too simple, discover its limitations and expand upon it, than to start with an overly complex reward model and be unable to identify why it did not work. Our experiences in designing the reward model for the power-management scenario are discussed further in Section 5.5.4.1.

The second difficulty in learning accurate policies was in parameterising the learning process. As discussed in Sections 5.2 and 5.3 we discovered quite early in our experiments that spatial and temporal dependencies could influence the accuracy of a learned policy, and that a very low learning rate was necessary to overcome these issues. This rate should be experimented with for each application

to maximise the rate at which the application learns. A related issue is that of identifying when learning can stop for a particular set of discrete states, and this again requires parameterisation. The appropriate learning time is completely dependent on the particular application, as the frequency with which values are encountered affects how often Q-values are updated and therefore how quickly those values change. This in turn affects how frequently policies should be compared to evaluate if they have ceased changing and learning has finished.

6.4 Future work

In the development of the approach described in this thesis a number of issues were identified that could be suitable for further investigation. This section outlines two key areas identified for future work: learning efficiency and increased automation of the KAFCA process.

6.4.1 Learning efficiency

This thesis addressed the challenge of learning accurate context definitions based on the most useful, available sensors in the run-time environment. Issues associated with learning time were not considered, although an evaluation of the trade-off between learning and improvements in context-definition accuracy was performed. This showed that reinforcement learning was slow to learn accurate policies, and required a ~50% increase in learning iterations for each refinement of context definitions. Although the learning time is application specific, this is clearly a significant issue for the deployment of KAFCA applications. We propose three possible directions for future work to address this concern: parallelising reinforcement learning of policies, reusing learned knowledge, and generalising about rewards for offline learning.

The current version of KAFCA serially refines the discrete states of individual sensors to define accurate context definitions, and then serially evaluates sensor combinations to identify the most useful set of sensors for an application. Each discrete-state refinement or sensor-combination evaluation requires a separate learning phase to learn a policy. The efficiency of the KAFCA process could be significantly improved if policies were learned in parallel. During learning the reward for an action is independent of the discrete state or context in which it is executed. It should be possible to parallelise the update of Q-values across many different sets of discrete states or contexts. Although this would require an increase in processing and memory requirements it would also greatly reduce the overall number of learning iterations required by KAFCA.

The current version of KAFCA learns a new policy for each set of discrete states during their

refinement and none of the knowledge from previously-learned policies is reused. Obviously there is a lot of similarity between policies as many of the discrete states do not change between refinements. Even in the case of discrete states that are split there are in theory only two actions that can be optimal for those states, i.e., the optimal actions of dissimilar neighbouring discrete states prior to splitting. It should be possible to reuse much of the learned knowledge of policies in order to reduce the learning required to identify a stable, accurate policy. It may even be possible to lead the application towards particular regions of the sensor space where learning is needed.

Another possible approach to addressing learning efficiency is to carry out some form of offline learning based on previously observed rewards. In Chapter 2 we discussed an approach proposed in (Zaidenberg et al., 2009) that created an offline world model by recording observed states, rewards and transitions. The world model was used to learn new policies without interacting with the real world. Apart from a number of issues regarding the completeness of their world model this approach would not work for KAFCA as its set of states is dynamic, i.e., the world model would be different for each sensor space, and each refinement of discrete states. Instead it could be possible to create an offline reward world by generalising about rewards given for actions in the real world. The sensor space of the reward model is fixed by the developer when they define the set of sensors needed to calculate rewards. Each reward is calculated for an action at a particular tuple in this sensor space. If these rewards could be generalised about, e.g., using probability distributions expressed on the sensor space or a clustering technique, then it would possible to learn policies by interacting with this reward world offline. This would obviously be a much quicker learning process than interacting with the real world. The accuracy of learned policies could be evaluated by observing their stability during a subsequent, short learning period in the real world.

6.4.2 Increased automation of the KAFCA process

This thesis addressed the limitations on context-definition accuracy and flexibility at run time that were introduced by dependency on expert knowledge. To this end it defined an approach to learning context definitions that was to a large extent independent of expert knowledge. A potential avenue of future work to further address the dependency on expert knowledge is in automation of the KAFCA process. The reward model and sensor-evaluation metric used by KAFCA are essential to guiding the reinforcement-learning and sensor-selection processes so that the application achieves its goals. These structures are necessarily dependent on expert knowledge to capture the goals of the application. The parameters that configure the KAFCA process are knowledge intensive but less application specific, so it may be possible to reduce the requirement for expert knowledge by further automating the KAFCA

process.

For example the number of discrete-state refinements to carry out for each sensor is currently defined by the developer. This requires the developer to understand and evaluate the tradeoff between learning and context-definition accuracy. It should be possible to apply some metric to application performance that measures this tradeoff and stops refinement when it does not yield sufficient improvement in context-definition accuracy. Another example of where the process could be automated is in detecting when to stop learning. The current method based on the stability of the policy requires the developer to define the number of iterations between stability tests and the required sequence of stable policies. Other approaches might be explored that used some standard means to evaluate stability, e.g., the rate of change of standard deviation among Q-values. The current implementation also depends on the developer to define the period between action executions. It would be interesting to examine how the decision-making process could be driven by events or significant changes in the environment. Obviously to automate the process these events or changes would have to be identified automatically.

Bibliography

Aamodt, A., & Plaza, E. (1994). Case-based reasoning; foundational issues, methodological variations, and system approaches. *Artificial Intelligence Communications 7*.

Abowd, G. D., Dey, A. K., Brown, P. J., Davies, N., Smith, M., & Steggles, P. (1999). Towards a better understanding of context and context-awareness. In *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, (pp. 304–307). London, UK: Springer-Verlag.

Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., & Verkamo, A. I. (1996). Fast discovery of association rules. (pp. 307–328).

Albinali, F., Davies, N., & Friday, A. (2007). Structural learning of activities from sparse datasets. In *PERCOM 07: Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications*, (pp. 221–228). Washington, DC, USA: IEEE Computer Society.

Ali, F. M., Lee, S. W., Bien, Z., & Mokhtari, M. (2008). Combined fuzzy state q-learning algorithm to predict context aware user activity under uncertainty in assistive environment. *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, ACIS International Conference on*, *0*, 57–62.

Atkin, M. S., & Cohen, P. R. (2000). Using simulation and critical points to define states in continuous search spaces. In *WSC 00: Proceedings of the 32nd Winter Simulation Conference*, (pp. 464–470). Society for Computer Simulation International.

Baldauf, M., Dustdar, S., & Rosenberg, F. (2007). A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, *2*(4), 263–277.

Banavar, G., & Bernstein, A. (2004). Challenges in design and software infrastructure for ubiquitous computing applications. *OecNews*, *62*, 180–203.

Bardram, J. E. (2005). The java context awareness framework (jcaf) : A service infrastructure and programming framework for context-aware applications. (pp. 98–115).

Baron, M. (2007). *Probability and Statistics for Computer Science students*. Chapman and Hall/CRC.

Barton, J., Zhai, S., & Cousins, S. (2006). Mobile phones will become the primary personal computing devices. *Mobile Computing Systems and Applications, 2006. WMCSA 06. Proceedings. 7th IEEE Workshop on*, (pp. 3–9).

Battestini, A., & Flanagan, J. A. (2005). Analysis and cluster based modelling and recognition of context in a mobile environment. In *Second International Workshop on Modelling and Retrieval of Context (MRC05)*.

Bell, G., & Dourish, P. (2007). Yesterdays tomorrows: Notes on ubiquitous computings dominant vision. *Personal and Ubiquitous Computing*, *11*(2), 133–143.

Benini, L., Bogliolo, A., & Micheli, G. D. (2000). A survey of design techniques for system-level dynamic power management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *8*(3), 299–316.

Biegel, G. (2004). *A Programming Model for Mobile, Context-Aware Applications*. Ph.D. thesis, University of Dublin, Trinity College.

Biegel, G., & Cahill, V. (2004). A framework for developing mobile, context-aware applications. In *IEEE International Conference on Pervasive Computing and Communications (PerCom 2004)*, (pp. 361–365). IEEE Computer Society.

Bolchini, C., Curino, C., Quintarelli, E., Schreiber, F. A., & Tanca, L. (2007). A data-oriented survey of context models. *SIGMOD Record*, *36*(4), 19–26.

Bowling, M., & Veloso, M. (2002). Multiagent learning using a variable learning rate. *Artificial Intelligence*, *136*, 215–250.

Brdiczka, O., Reignier, P., Crowley, J. L., Vaufreydaz, D., & Maisonnasse, J. (2006). Deterministic and probabilistic implementation of context. In *Pervasive Computing and Communications Workshops*.

Brown, P. J. (1996). The stick-e document: a framework for creating context-aware applications. In *Proceedings of Electronic Publishing (EP96)*, (pp. 259–272).

Callan, R. (2003). *Artificial Intelligence*. Palgrave Macmillan.

Chang, K.-H., Chen, M., & Canny, J. (2007). Tracking free-weight exercises. *UbiComp 2007: Ubiquitous Computing*, (pp. 19–37).

Charniak, E. (1991). Bayesian networks without tears. *AI Magazine*, (pp. 50–63).

Chen, C., & Helal, S. (2008). Sifting through the jungle of sensor standards. *Pervasive Computing, IEEE*, *7*(4), 84–88.

Clarkson, B., & Pentland, A. (1998). Extracting context from environmental audio. *Wearable Computers, 1998. Digest of Papers. Second International Symposium on*, (pp. 154–155).

Cooper, G. F., & Herskovits, E. (1992). A bayesian method for the induction of probabilistic networks from data. *Machine Learning*, *9*(4), 309–347.

da Costa, C., Yamin, A., & Geyer, C. (2008). Toward a general software infrastructure for ubiquitous computing. *Pervasive Computing, IEEE*, *7*(1), 64–73.

Dey, A., Salber, D., & Abowd, G. (2001). A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications.

Dey, A. K., & Abowd, G. D. (2000). Towards a better understanding of context and context-awareness. In *Workshop on the What, Who, Where, When and How of Context-awareness*. New York: ACM.

Flanagan, J. A. (2005a). Context awareness in a mobile device: ontologies versus unsupervise/supervised learning. In *International and Interdisciplinary Conference on Adaptive Knowledge Representation and Reasoning (AKRR05)*.

Flanagan, J. A. (2005b). Unsupervised clustering of context data and learning user requirements for a mobile device. In *Modeling and Using Context, 5th International and Interdisciplinary Conference, CONTEXT 2005*, (pp. 155–168).

Gartner (2009). Context-aware computing gains momentum.
URL `http://gartner.com/DisplayDocument?doc_cd=163256`

Gellersen, H. W., Schmidt, A., & Beigl, M. (2002). Multi-sensor context-awareness in mobile devices and smart artifacts. *Mobile Networks and Applications*, *7*(5), 341–351.

Gonzalez, A. J., & Ahlers, R. (1998). Context-based representation of intelligent behavior in training simulations. *Trans. Soc. Comput. Simul. Int.*, *15*(4), 153–166.

Gu, T., Pung, H., & Zhang, D. (2004). A middleware for building context-aware mobile services.

Harris, C. (2007). *Context-Aware Power Management*. Ph.D. thesis, University of Dublin, Trinity College.

Harris, C., & Cahill, V. (2005a). Exploiting user behaviour for context-aware power management. In *International Conference On Wireless and Mobile Computing, Networking and Communications*, (pp. 122–130). IEEE Computer Society.

Harris, C., & Cahill, V. (2005b). Power management for stationary machines in a pervasive computing environment. In 38^{th} *Annual Hawaii International Conference on System Sciences (HICSS05)*, (p. 285a). IEEE Computer Society.

Henricksen, K., Indulska, J., & Rakotonirainy, A. (2006). Using context and preferences to implement self-adapting pervasive computing applications. *Software Practice and Experience*, *36*(11-12), 1307–1330.

Himberg, J., Flanagan, J. A., & Mantyjarvi, J. (2003). Towards context awareness using symbol clustering map. In *In Proc. Workshop for Self-Organizing Maps 2003 (WSOM2003)*, (pp. 249–254).

Hofer, T., Schwinger, W., Pichler, M., Leonhartsberger, G., Altmann, J., & Retschitzegger, W. (2003). Context-awareness on mobile devices - the hydrogen approach. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, (pp. 10 pp.+).

Hong, J. I., & Landay, J. A. (2004). An architecture for privacy-sensitive ubiquitous computing. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, (pp. 177–189). New York, NY, USA: ACM.

Hull, R., Neaves, P., & Bedford-Roberts, J. (1997). Towards situated computing. (pp. 146–153).

Intel (2006). Opencv.
URL `http://sourceforge.net/projects/opencvlibrary/`

Intel (2007). Integrated performance primitives.
URL `http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm`

Jonsson, M., Werle, P., & Jansson, C. G. (2003). Context shadow: An infrastructure for context aware computing. In *Proceedings of Artificial Intelligence in Mobile Systems*.

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, *4*, 237–285.

Kofod-Petersen, A. (2006). Challenges in case-based reasoning for context awareness in ambient intelligent systems. In M. Minor (Ed.) *8th European Conference on Case-Based Reasoning, Workshop Proceedings*, (pp. 287–299). Oudeniz/Fethiye, Turkey.

Kofod-Petersen, A., & Aamodt, A. (2006). Contextualised ambient intelligence through case-based reasoning. In *The 8th European Conference on Case-Based Reasoning (ECCBR 2006)*. Oudeniz/Fethiye, Turkey.

Kohonen, T. (2001). *Self-Organising Maps*. Springer.

Kolodner, J. (1993). *Case-Based Reasoning*. Morgan Kaufmann Publishers.

Korpipaa, P., Koskinen, M., Peltola, J., Makela, S.-M., & Seppanen, T. (2003). Bayesian approach to sensor-based context awareness. *Personal Ubiquitous Comput.*, *7*(2), 113–124.

Korpipaa, P., Malm, E.-J., Salminen, I., Rantakokko, T., Kyllonen, V., & Kansala, I. (2005). Context management for end user development of context-aware applications. In *MDM 05: Proceedings of the 6th international conference on Mobile data management*, (pp. 304–308). New York, NY, USA: ACM.

Krause, A., Smailagic, A., & Siewiorek, D. P. (2006). Context-aware mobile computing: Learning context-dependent personal preferences from a wearable sensor array. *IEEE Transactions on Mobile Computing*, *5*(2), 113–127.

Leahu, L., Sengers, P., & Mateas, M. (2008). Interactionist ai and the promise of ubicomp, or, how to put your box in the world without putting the world in your box. In *UbiComp 08: Proceedings of the 10th international conference on Ubiquitous computing*, (pp. 134–143). New York, NY, USA: ACM.

Ma, L., Smith, D., & Milner, B. (2003). Environmental noise classification for context-aware applications. *Lecture Notes in Computer Science : Database and Expert Systems Applications*, (pp. 360–370).

Ma, T., Kim, Y.-D., Ma, Q., Tang, M., & Zhou, W. (2005). Context-aware implementation based on cbr for smart home. In *Wireless And Mobile Computing, Networking And Communications, 2005. (WiMob2005), IEEE International Conference on*, vol. 4, (pp. 112–115 Vol. 4).

Meier, R., & Cahill, V. (2003). Exploiting proximity in event-based middleware for collaborative mobile applications. (pp. 285–296).

Mikalsen, M., & Kofod-petersen, A. (2005). Representing and reasoning about context in a mobile environment. *Revue Intelligence Artificielle (RIA)*, *19*, 479–498.

Modahl, M., Agarwalla, B., Saponas, S., Abowd, G., & Ramachandran, U. (2005). Ubiqstack: a taxonomy for a ubiquitous computing software stack. *Personal Ubiquitous Comput.*, *10*(1), 21–27.

Moore, D. (1999). *The Basic Practice of Statistics*. New York, NY, USA: W. H. Freeman & Co.

Moore, P., Hu, B., Zhu, X., Campbell, W., & Ratcliffe, M. (2007). A survey of context modeling for pervasive cooperative learning. *Information Technologies and Applications in Education ISITAE07*, (pp. K5–1–K5–6).

Muhlenbrock, M., Brdiczka, O., Snowdon, D., & Meunier, J. L. (2004). Learning to detect user activity and availability from a variety of sensor data. In *PERCOM 04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*. Washington, DC, USA: IEEE Computer Society.

National Institute of Standards and Technology (2005). Ieee 1451: Smart transducer interface standard.
URL `http://ieee1451.nist.gov/`

Nehaniv, C. L. (1999). Meaning for observers and agents. In *Internation symposium on Intelligent Control/Intelligent Systems and Semiotics*. Cambridge.

Ogden, C. K., & Richards, I. A. (1923). *The Meaning of Meaning*. University of Cambridge.

Open Geospatial Consortium (2000). Sensor model language (sensorml).
URL `http://www.opengeospatial.org/standards/sensorml`

Padovitz, A., Loke, S. W., Zaslavsky, A., & Bartolini, C. (2005). An approach to data fusion for context awareness. In *Fifth International Conference on Modelling and Using Context*, (pp. 353–367).

Park, H.-S., Yoo, J.-O., & Cho, S.-B. (2006). A context-aware music recommendation system using fuzzy bayesian networks with utility theory. *Fuzzy Systems and Knowledge Discovery*, (pp. 970–979).

Pascoe, J. (1998). Adding generic contextual capabilities to wearable computers. *Wearable Computers, IEEE International Symposium*.

Polani, D., Martinetz, T., & Kim, J. (2001). An information-theoretic approach for the quantification of relevance. In *European Conference on Artificial Life (ECAL)*. Prague.

Prekop, P., & Burnett, M. (2003). Activities, context and ubiquitous computing. *Special Issue on Ubiquitous Computing Computer Communications*.

Ranganathan, A., & Campbell, R. (2003). A middleware for context-aware agents in ubiquitous computing environments. (p. 998).

Russell, S., & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Ryan, N. S., Pascoe, J., & Morse, D. R. (1998). Enhanced reality fieldwork: the context-aware archaeological assistant. In V. Gaffney, M. van Leusen, & S. Exxon (Eds.) *Computer Applications in Archaeology 1997*, British Archaeological Reports. Oxford: Tempus Reparatum.

Sadeh, N., Gandon, F., & Kwon, O. B. (2005). Ambient intelligence: The mycampus experience. Tech. rep., School of Computer Science, Carnegie Mellon University.

Salkham, A., Cunningham, R., Garg, A., & Cahill, V. (2008). A collaborative reinforcement learning approach to urban traffic control optimization. *Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM International Conference on*, *2*, 560–566.

Satyanarayanan, M. (2001). Pervasive computing: vision and challenges. *IEEE Personal Communications*, *8*(4), 10–17.

Schilit, B. N., & Theimer, M. M. (1994). Disseminating active map information to mobile hosts. *Network, IEEE*, *8*(5), 22–32.

Schmidt, A., Aidoo, K. A., Takaluoma, A., Tuomela, U., Laerhoven, K. V., & Velde, W. V. D. (1999). Advanced interaction in context. In *In Proceedings of First International Symposium on Handheld and Ubiquitous Computing*, (pp. 89–101). Springer Verlag.

Schmidt, A., Strohbach, M., Laerhoven, K. V., Friday, A., & w. Gellersen, H. (2002). Context acquisition based on load sensing. In *In Proceedings of UbiComp: Ubiquitous Computing*, (pp. 333–350). Springer Verlag.

Senart, A., Cunningham, R., Bouroche, M., Connor, N. O., Reynolds, V., & Cahill, V. (2006). MoCoA: Customisable middleware for context-aware mobile applications. In *8th International Symposium on Distributed Objects and Applications (DOA 2006)*, vol. 4276 of *Lecture Notes in Computer Science*, (pp. 1722–1738). Springer Verlag.

Simunic, T., Benini, L., Glynn, P. W., & Micheli, G. D. (2000). Dynamic power management for portable systems. In *Mobile Computing and Networking*, (pp. 11–19).

Smith, D., Ma, L., & Ryan, N. (2006). Acoustic environment as an indicator of social and physical context. *Personal and Ubiquitous Computing*, *10*(4), 241–254.

Strang, T., & Linnhoff-Popien, C. (2004). A context modeling survey. In *Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004 - The Sixth International Conference on Ubiquitous Computing*.

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *In Proceedings of the Seventh International Conference on Machine Learning*, (pp. 216–224). Morgan Kaufmann.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

Tapia, E. M., Choudhury, T., & Philipose, M. (2006). Building reliable activity models using hierarchical shrinkage and mined ontology. In *The 4th International Conference on Pervasive Computing*. Springer Berlin / Heidelberg.

Tsang, S. L. (2009). *Supporting Personalised Recommendations in Context-aware Applications*. Ph.D. thesis, University of Dublin, Trinity College.

Tsang, S. L., & Clarke, S. (2007). Mining user models for effective adaptation of context-aware applications. In *IPC 07: Proceedings of the The 2007 International Conference on Intelligent Pervasive Computing*, (pp. 178–187). Washington, DC, USA: IEEE Computer Society.

Van Laerhoven, K. (2001). Combining the self-organizing map and k-means clustering for on-line classification of sensor data. (pp. 464–469).

Van Laerhoven, K., & Cakmakci, O. (2000). What shall we teach our pants? *Wearable Computers, 2000. The Fourth International Symposium on*, (pp. 77–83).

Veeramachaneni, S., Sarkar, P., & Nagy, G. (2005). Modeling context as statistical dependence. (pp. 515–528).

Watkins, C. J., & Dayan, P. (1992). Technical note: Q-learning. *Machine Learning*, *8*(3), 279–292.

Weiser, M. (1991). The computer for the 21st century. *Scientific American*, *265*(3), 66–75.

Weiser, M. (1993). Some computer science issues in ubiquitous computing. *Commun. ACM*, *36*(7), 75–84.

Wittgenstein, L. (1968). *Philosophical Investigations*. Basil Blackwell.

Wolfe, M. (2003). Smart couch report. Tech. rep., Department of Computer Science, Trinity College Dublin.

Zadeh, L. A. (1965). Fuzzy sets. *Information and Control*, *8*(3), 338–353.

Zaidenberg, S., Reignier, P., & Crowley, J. (2009). Reinforcement learning of context models for a ubiquitous personal assistant. (pp. 254–264).

Zimmermann, A. (2003). Context-awareness in user modelling: Requirements analysis for a case-based reasoning application.