Technical Report

UCAM-CL-TR-620
ISSN 1476-2986

Number 620

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Middleware support for context-awareness in distributed sensor-driven systems

Eleftheria Katsiri

February 2005

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted January 2005 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Clare College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Abstract

Context-awareness concerns the ability of computing devices to detect, interpret and respond to aspects of the user's local environment. *Sentient Computing* is a sensor-driven programming paradigm which maintains an event-based, dynamic model of the environment which can be used by applications in order to drive changes in their behaviour, thus achieving context-awareness. However, primitive events, especially those arising from sensors, e.g., that a user is at position (x, y, z), are too low-level to be meaningful to applications. Existing models for creating higher-level, more meaningful events, from low-level events, are insufficient to capture the user's intuition about abstract system state. Furthermore, there is a strong need for user-centred application development, without undue programming overhead. Applications need to be created dynamically and remain functional independently of the distributed nature and heterogeneity of sensor-driven systems, even while the user is mobile. Both issues combined necessitate an alternative model for developing applications in a real-time, distributed sensor-driven environment such as Sentient Computing.

This dissertation describes the design and implementation of the SCAFOS framework. SCAFOS has two novel aspects. Firstly, it provides powerful tools for inferring *abstract knowledge* from low-level, *concrete* knowledge, verifying its correctness and estimating its likelihood. Such tools include Hidden Markov Models, a Bayesian Classifier, Temporal First-Order Logic, the theorem prover SPASS and the production system CLIPS. Secondly, SCAFOS provides support for simple application development through the XML-based SCALA language. By introducing the new concept of a generalised event, an *abstract event*, defined as a notification of changes in abstract system state, expressiveness compatible with human intuition is achieved when using SCALA. The applications that are created through SCALA are automatically integrated and operate seamlessly in the various heterogeneous components of the context-aware environment even while the user is mobile or when new entities or other applications are added or removed in SCAFOS.

To my parents Nick and Alex and my brother Dimitris, for a lifetime of love and support.

Acknowledgements

I am indebted to Professor Andy Hopper who has made it possible for me to pursue a PhD degree and provided invaluable mentoring and overall guidance, yet promoted the freedom to explore.

I am deeply grateful to Dr. Alan Mycroft without whom this thesis would not have materialised for his sterling performance as my supervisor and for shaping my research and education in the field.

I would like to offer my sincere thanks to Professor Mike Gordon and Dr. Jean Bacon for some enduring guidance and hearty encouragement. Many thanks go also to Dr. Wassel, for encouragement and support.

Many thanks to Clare College for their rigorous support and tutoring throughout this effort.

I would like to thank all my friends in the LCE past and present, and in particularly Pablo, Hani, Ernst, Kasim, Jenni, John, Jamie, Wez, Karen and Inaki who make working in the LCE so enjoyable. Special thanks to Jamie, Wez, Karen and Jenny for proof-reading my thesis.

This work has been made possible through the generous support of the University of Cambridge, Clare College and AT&T Research (Cambridge) through a Domestic Research Studentship, a Steel Pressed Award and a CASE award respectively. I would also like to thank the Engineering Department for additional funding.

Publications

This dissertation is based in part on work included in the following publications:

E. Katsiri, J. Bacon and A. Mycroft. An Extended Publish/Subscribe Protocol for Transparent Subscriptions to Distributed Abstract State in Sensor-Driven Systems using Abstract Events. In *Proceedings of the 3rd International Workshop on Distributed Event-Based Systems (DEBS04)*, Edinburgh, UK, May 2004.

E. Katsiri and A. Mycroft. Knowledge Representation and Scalable Abstract Reasoning for Sentient Computing Using First-Order Logic. In *Proceedings of the 1st Workshop on Challenges and Novel Applications for Automated Reasoning, in conjunction with CADE-19*, Miami Beach, FL, July 2003.

E. Katsiri. Principles of Context Inferences. In *Ubicomp2002 Adjunct Proceedings*, Gothenbugh, Sweden, Sept. 2002.

E. Katsiri. A Context-Aware Notification Service. In *Proceedings of the 1st European Workshop of Location Based Services*, London, UK, Sept. 2002.

D. Lopez de Ipina and E. Katsiri. An ECA Rule-Matching Service for Simpler Development of Reactive Applications. In *Middleware 2001 Adjunct Proceedings, published at IEEE Distributed Systems Online*, Nov. 2001.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

επιστήμη εστί δόξα μετά λόγου αληθές.
(Knowledge is true belief plus an account of Logos[1]) — Plato (Theaetetus 201d)

Context-awareness [96] concerns the ability of computing devices to detect, interpret and respond to aspects of the user's local environment. Its goal is to enhance computer systems with a sense of the real world and make them know as much as the user about the aspects of the environment relevant to their application. We refer to environments enhanced with context-awareness as *context-aware environments* and to the applications that operate in such environments as *context-aware applications*.

Context-awareness is orthogonal to programming paradigms such as Ubiquitous Computing [108] and Sentient Computing [44], which aim to optimise the service offered to the user through applications. Such applications follow a common modus operandi: the user specifies a set of requirements in terms of abstract, high-level context as well as the service to be delivered, once the specified context has occurred. The application receives notifications from the Sentient system whenever the specified context happens, and it executes accordingly the specified action. For example, the user may ask to be reminded to return John's book when located in the same room as John.

Although sensor-driven systems are physically distributed and employ events for acquiring and communicating awareness of their surrounding environments, they bear significant differences to traditional event-based distributed systems. The events that are produced by sensors are too low-level to be meaningful to the applications. For example, the event that a user is at position (x, y, z) is not directly usable by an application that is interested in determining when a user is in close proximity to a PC. Furthermore, sensor-derived events do not convey *negative* information, i.e., information that something has not happened. For example, although an event that notifies of a fire can be easily produced by a fire alarm, unless such an event occurs, there is no information about the absence of fire. Another issue of sensor-driven systems is that these typically consist of several heterogeneous components that differ significantly in terms of the properties of the instrumenting technology, e.g., the accuracy of the location system, the number of users that are known in the domain and the specific topology of the domain. This has a huge impact on application development. A user that moves between domains may need to be able to locate the *closest, empty meeting room* in each of the visited domains, without recompiling the application that delivers this information with each transition to a new domain. Furthermore, users may need to develop new applications ad hoc, and similarly cancel their operation in real-time. This means that application

[1] *The Greek word Logos (traditionally meaning word, thought, principle, or speech) has been used among both philosophers and theologians. In most of its usages, Logos is marked by two main distinctions - the first dealing with human reason and the second with universal intelligence i.e., the Divine. The word was used by Heraclitus, one of the pre-eminent Pre-Socratic Greek philosophers, to describe human knowledge and the inherent order in the universe, a background to the essential change which characterises day-to-day life. By the time of Socrates, Plato, and Aristotle, Logos was the term used to describe the faculty of human reason and the knowledge men had of the world and of each other. The Stoics understood Logos as the animating power of the universe, which further influenced how this word was associated with the Divine.*

development needs to be dynamic and without undue programming effort. Lastly, Ubiquitous Computing advocates that the computing infrastructure that enables awareness should be *invisible* from the user. This necessitates a separation of concerns between the way users define contextual situations in order to build applications and the way context is managed by the sensor-driven system.

Existing techniques for modelling distributed, event-based systems, such as *event composition*, are not adequate for modelling context-aware systems, as they cannot capture certain human intuition involving state. Furthermore, these techniques do not cater for *transparency*, i.e., hiding the effect of the heterogeneity of sensor-driven components from the modelling mechanism. On the other hand, existing models for context-awareness such as SPIRIT [41] are not programmable. As far as applications are concerned, context-aware application development has been ad hoc, often overwhelming to the user, depriving him of any control over the desired application behaviour. This introduces an imperative need for tools that aid context-aware application development in a way that the desired application functionality is determined by the user, *dynamically*, while being mobile. All the above reasons necessitate an alternative model for context-awareness in sensor-driven systems.

This dissertation describes the design and implementation of the SCAFOS framework. SCAFOS achieves two principal goals. Firstly, it implements a model for context-awareness for sensor-driven systems, which satisfies the above modelling requirements. The proposed model is *standardised, user-centred, distributable, portable, dynamically extensible* and provides an *invisible separation of concerns between knowledge acquisition, integrated knowledge management and knowledge usage by applications*, thus ensuring a natural interface to the users, abiding by the principles of Ubiquitous Computing. Secondly, SCAFOS promotes application development without undue programming overhead by providing a set of tools and services for the creation, deployment and operation of context-aware applications in *distributed, dynamic, extensible, heterogeneous, context-aware environments*. The language part of SCAFOS is called SCALA; it is XML-based, and it allows for top-down, user-controlled application development. Applications operating within this framework are created automatically from user specifications, and benefit from automatic integration with little development overhead; their seamless operation is guaranteed as the user moves, even when the model is extended dynamically.

1.1 Vision

The user in the near future is perceived to be mobile and to move through several *heterogeneous* indoor and outdoor *context-aware environments*. The user carries with him a Personal Digital Assistant (PDA), through which it is possible to interact directly with the physical environment, specifying the action that is to take place whenever a *situation* of interest occurs. Such situations of interest can have the form of queries, e.g., querying the location of *the closest wireless hot spot that guarantees a connection speed over a certain threshold*. The user can also define applications that remain functional for a specific period of time. For example, the user may ask to be notified whenever the closest system administrator becomes available within the next 10 minutes, or he may request that *all personal messages be withheld whenever he is talking with colleagues with whom he spends on average less than 5% of his weekly time, and that this application be effective for the rest of the working day and no longer than that*.

Apart from the predefined set of requirements, ad hoc requirements can also be issued during the user's daily routine, such as *notify me by SMS when the boss arrives in the office* or even *show me a map to the closest cinema that shows the latest movie by Almodovar*. Such requests are generated by typing a command on a Personal Digital Assistant (PDA), and they take effect immediately, unless otherwise specified.

A user's environment typically involves a home, an office with various departments, a car, malls, recreational areas such as cinemas, etc. Such environments are dynamic, and even their topology may evolve continually. For example, a multi-departmental institution may expand by modifying a closed-space topology to a rent-a-desk open space topology. By analysing what is required to materialise this

vision, the following requirements for context-awareness have been defined.

Formal Modelling. A model that encompasses the envisioned behaviour of context-aware systems and their users is necessary. Such a model promotes the understanding of the issues involved in the context-aware paradigm, unveils the capabilities and potential of each existing solution, and offers a platform where development efforts can be standardised and replicated effort can be avoided. Although several models already exist for traditional event-based distributed systems, there is no adequate model available for context-aware systems.

User-Centred Model and Application Development. In a model for context-aware behaviour such as the one envisioned above, users need to be able to determine which actions will take place under which conditions and what effect such actions will have on their interruptibility - i.e., when it is appropriate to disturb them. This means that context-aware application development should be simple enough to be undertaken by the user, and applications should become effective (and similarly ineffective) in real-time. Such applications need to operate seamlessly as the user moves from one context-aware environment to another. They also need to remain operational under the specified temporal constraints, even when other users develop other applications dynamically or when the entities (users, topology) in the surrounding environment change.

Invisibility and Transparency. The sensor and computing infrastructure that enables context-awareness needs to be invisible to the user, thus providing a natural interface for developing applications. *Invisibility* advocates that the application logic is compatible with human intuition. *Transparency* guarantees that the application logic will be functional in all distributed components of a context-aware environment, independent of the entities (users, objects, topology) that exist in that component.

Expressiveness. In order to reason with the model and develop application logic, there needs to be available language support. *Expressiveness* guarantees that such a language is powerful enough for expressing all human intuition.

Real-Time Constraints. As mentioned before, applications are developed dynamically, while the system is running and without additional recompilation. This introduces real-time constraints that need to be adhered to.

1.2 A Philosophical View on Context, Knowledge and Events

This section aims to give a brief philosophical background of some of the key concepts in the conceptual framework described in this dissertation. This is meant to demonstrate the historical continuity in the intellectual challenge that such concepts raise, as well as the similarities and the differences between the philosophical approach and the approach advocated by the current trends in computer science and adopted in this thesis. The references given here are indicative, and are really only used as an index into the wider literature.

Context. *Context* is a term which is widely used in computer science, and various definitions for context exist in the literature. Various areas of computer science, such as natural language processing, have investigated this concept over the last 40 years to relate information processing and communication to aspects of the situations in which such processing occurs. Schmidt et al [98] have given the following definition for context: *"Context is what surrounds us and gives meaning to something else."* Dey talks about a different definition of context [29]. *"Context is any information that can be used to characterise*

the situation of an entity. An entity is a person, a place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves." This definition is more appropriate for describing interactions in context-aware applications.

This thesis takes a step back and identifies the similarities that exist between the above definitions of context and the concept of *knowledge* as this has been used in computer science and particularly in Artificial Intelligence. It argues that *context is knowledge of the state of entities in the surrounding world* and it considers *events* to be *changes of such state*. But what exactly is knowledge?

Knowledge. According to the *Dictionary of Philosophy* [35], knowledge is a combination the following: (i) knowledge that (factual knowledge), (ii) knowledge how (practical knowledge), and (iii) knowledge of people, places, and things (knowledge by acquaintance).

Plato in *Meno* [83] offers a distinction between true belief and knowledge (97d-98b). Knowledge, Plato says, is tethered in a way that true belief is not. This view, which seems to suggest that knowledge is justified true belief, is taken up again in the *Theaetetus* [85], where Plato suggests that *knowledge is true belief plus an account of logos* (201d). In the *Republic* [84], Plato identifies knowledge with infallibility or certainty by claiming that, knowledge is infallible, while belief is fallible (Republic 477e). In the *Republic* Book V, Plato addresses a version of the question pertaining to the extent of knowledge. There he distinguishes between knowledge at one extreme and ignorance at the other, and he roughly identifies an intermediate state as belief. Each of these states of mind, Plato says, has an object. The object of knowledge is what is or exists; the object of ignorance is what does not exist; and the object of belief is some intermediate entity, often taken as the sensible physical world of objects and their qualities (Republic 508d-e; Cratylus [82] 440a-d). What truly exists for Plato are unchanging Forms, and it is these which he indicates as the true objects of knowledge.

Aristotle elaborates on the Platonic view of knowledge by distinguishing between *perceptual* and *propositional knowledge*. The former is perceived by the senses and the latter is knowledge of facts. Perceptual knowledge is discussed in *De Anima* [2], as knowledge that is perceived through the senses and therefore is subject to error. However, in the *Posterior Analytics* [3], Aristotle takes this view of knowledge even further and discusses a special form of knowledge, *scientific knowledge* as a form of *logical deduction*. A science, as Aristotle understands it, is to be thought of as a group of theorems each of which is proved in a demonstrative syllogism. In the first instance, a demonstrative syllogism in science S is a syllogistic argument whose premises are first principles of S. These first principles, in turn, must be true, primary, immediate, better known than and prior to the conclusion, which is further related to them as effect to cause (Posterior Analytics 71b21-22). Knowledge of the conclusion requires knowledge of the first principles, but not conversely. The science can be extended by taking theorems proved from first principles as premises in additional demonstrative syllogisms for further conclusions. A person who carries through all these syllogisms with relevant understanding has knowledge of all of the theorems.

Events. Events form an apparently distinct kind, different from things like people, planets and books. Many events are changes, for example, human bodies being first alive and then dead or things being first hot and then cold. But this may not define events fully. Events can be used in order to distinguish intrinsic changes, like dying, from some relational ones, like being orphaned. Second, events that begin or end things, like the Big Bang and other explosions, cannot be changes in themselves and may not, if nothing precedes or survives them, be changes in anything else. The difference between things and events, whether changes or not, may be that things keep a full identity over time, which events lack. First, some events may be instantaneous and lack any identity over time. Second, temporally extended events are deprived of full identity over time by their temporal parts, like a speech's spoken words, which stop them ever being wholly present at an instant; whereas people and other things have no temporal parts and are wholly present at every instant of their lives. This full identity over time will then distinguish

one thing changing from successive things having different properties, thus explaining why only things can change and why changes, being events, are not things.

The Interpretation of this Dissertation. The model proposed in this dissertation abides by both the above Platonic and Aristotelian views about knowledge through the following interpretation: knowledge is factual (predicate) and it is considered as the state of logical entities, which is either perceived, logically deduced or inferred from perceived or other deduced or inferred knowledge. Perceptual knowledge is a key component of sensor-driven systems, where perception can be seen as sensing, carried out by sensors. Perceptual knowledge is fallible. Indeed, the certainty of knowledge that is produced by sensors depends on the reliability of the sensor technology, thus sensed knowledge is in fact belief. Although this is not directly addressed in this thesis, there is active research attempting to model sensing accuracy with confidence levels, thus modelling the degree of belief. The modelling approach taken in this thesis employs similar principles (confidence levels and likelihood estimation) in order to model higher-level belief. It is worth noting that except for sections of the dissertation in which uncertainty is explicitly discussed, it can be assumed that uncertainty is small enough to be ignored. In this case the term knowledge will be used to mean both knowledge and belief, unless otherwise stated.

Knowledge in this thesis is deduced from perceived knowledge (sensor data) using temporal first-order logic (TFOL) and inferred using statistical probabilistic methodologies such as Hidden Markov Models and Bayesian Prediction. Perceived knowledge is referred to as *concrete* and deduced or inferred knowledge as *abstract*. Abstract knowledge is encapsulated as instances (in the Prolog sense) of abstract predicates. Lack of knowledge is considered as *ignorance*. Ignorance is a key issue in this dissertation, as it is not addressable by existing distributed system models. *Uncertainty* is another key issue which is inherent in context-aware systems. Wherever uncertainty plays an important role (Chapters 3 and 4), knowledge is equivalent to *belief*. Otherwise, the term knowledge will be used to mean both knowledge and belief, concrete and abstract, unless otherwise stated. Furthermore, these concepts are considered in the present, the past and the future, through current, historical and predicted data.

1.3 Sentient Computing and Sensor-Driven Systems

The research described in this dissertation is based on the *Sentient Computing* paradigm. Sentient Computing is a programming paradigm for context-awareness, which employs sensors, distributed through the environment, in order to create and maintain a detailed model of the real world and make it available to applications. This section elaborates on the requirements of Section 1.1 and focuses on their satisfiability in the context of Sentient Computing. Sensor-driven systems bear similarities with sensor networks, which are an emerging area of research.

A key characteristic of such systems is that events in Sentient Computing are not meaningful. Rather, they are instances of the state of an entity, as this is captured by the sensor technology. Furthermore, events in sensor-driven systems only convey positive knowledge. They cannot represent knowledge about something that has not occurred or has stopped occurring. Furthermore, the *semantic mapping* between concrete knowledge produced by sensors and abstract knowledge that is queried by the application layer is incomplete. *Uncertainty* is also an issue that affects the degree of abstract belief that is maintained by the system and queries by applications. Uncertainty can be introduced by various factors. Sensor *occlusion* may prevent an event from being generated. *Statistical error* in the location technology may cause the generation of events that are only approximations of the actual monitored entity property. *Node failure* is a common failure, especially in sensor networks.

Another key issue is that of *scalability*. Scalable computation and communication is crucial in processing the high update rate that is involved when reasoning with sensor updates, while maintaining a near real-time response. When scaling context-aware systems to a large number of physically distributed

components, which contain a large number of users that issue a large number of application requirements, maintaining an acceptable response time becomes significantly harder.

Finally, the *heterogeneity* of the technology and the entities involved in each distributed sensor-driven component, along with the dynamic evolution of such systems, often hinders the dynamic creation and the continuous, seamless operability of created applications. This is only aggravated by user mobility.

The satisfiability of the requirements of Section 1.1, given the above issues, is investigated and addressed in this dissertation.

1.4 A Conceptual Framework

This section describes the conceptual framework proposed in this dissertation and discusses the key decisions involved in its design and implementation. The main components are portrayed in Figure 1.1. The main application programmable interfaces (APIs) between the conceptual components will be discussed in detail in the following chapters and are only outlined here.

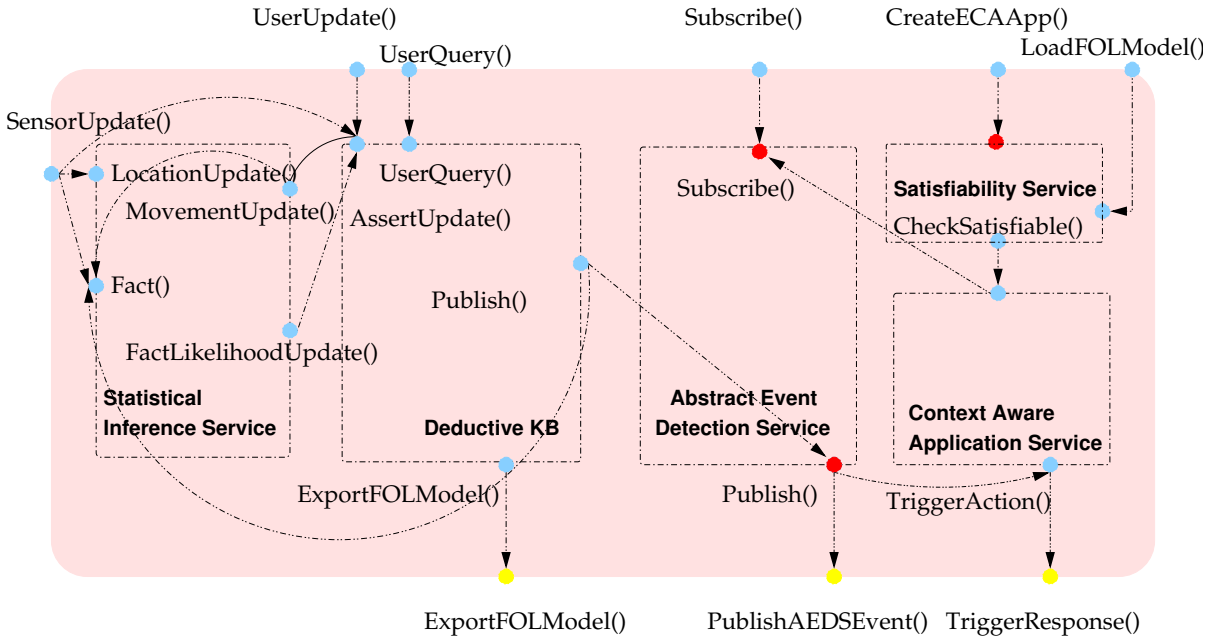


Figure 1.1. The SCAFOS conceptual framework.

The framework promotes user-driven context-aware application development. The user creates a specification for a context-aware application using an XML-based high-level language called SCALA. This is modelled by the *CreateECAApp()* interface. SCALA is discussed extensively in Chapter 12. Each SCALA definition consists of three components. An *ECAAS specification* (an application definition), an *AESL definition* (a definition of the abstract predicate and the rules by which this will be generated from primitive events) and an *AEFSL definition* (filter) (an assignment of constant values to the attributes of the abstract predicate of the AESL definition). The ECAAS specification defines the desired functionality of the context-aware application in an *enhanced* Event-Condition-Action (ECA) format, which defines that a particular system action (A) will be triggered whenever it is determined through the reception of an *abstract event* (E) that a condition (C) has been satisfied in the context-aware environment. Abstract Events are discussed in Chapter 8. This format is enhanced with *else* and *scope* statements. The *else* statement defines alternative actions to be triggered when the condition does not hold. The *scope* statement defines

temporal constraints on the application functionality, i.e., the length of time it remains in effect. Both the *else* and *scope* statements are discussed in Chapter 12.

Abstract knowledge can be inferred from concrete knowledge by means of Hidden Markov Models (HMMs) and Temporal First-Order Logic (TFOL). Chapter 4 presents a movement recognition methodology, based on HMMs. This process is undertaken by the Statistical Inferencing Service component of Figure 1.1. The algorithm for deducing the abstract knowledge predicates from concrete ones is defined by the user using an *AESL definition* and a filter, defined in the languages AESL and AEFSL respectively. Both languages are based on TFOL and are designed with computational efficiency in mind. Chapter 12 discusses the design principles behind them.

The AESL definition is checked for correctness and semantic compatibility with the various heterogeneous distributed components of the sensor-driven systems, using the Satisfiability Service through the *CheckSatisfiable()* interface (Chapter 9). Each component model is loaded in the Service through the *LoadFOLModel()* interface. This service will raise an exception if the AESL definition is not logically satisfiable. This may be due to incorrect usage of the underlying model, missing predicates, incorrect syntax or a mismatch between the application requirements and the technological properties such as location-sensing precision. This service also detects errors that can occur from violating logical constraints such as those derived from the functional operation of location predicates. For example, the situation where the C expert is typing while at the same time the system administrator is having a coffee is impossible if the system administrator and the C expert are the same person.

The combined AESL and AEFSL definitions are compiled into an *implementation process* similar to an execution *thread*, which uses the event-driven paradigm in order to communicate with the loosely coupled components of the framework that implement the reasoning required for creating the application. Although the event-driven paradigm is appropriate for this process, basic events, especially those arising from sensors, may be too low-level to be meaningful to users. For this reason, substantial research into *event composition* [6, 7, 63, 81, 102], i.e., the combination of primitive events into composite events by applying a set of *composition operators*, has been pursued by several groups. This bottom-up approach, although valuable, is insufficient to satisfy all the queries users might wish to make. In particular, composition using finite state machines cannot capture the user's intuition about abstract system state or express concepts that involve *negation*, such as *"empty room"*. This means that they cannot model *ignorance* as discussed in Section 1.2. At the same time, event composition is dependent on the specific implementation domain, which hinders *semantic transparency*. This necessitates an alternative model for querying, and subscribing transparently to, distributed state in a real-time, ubiquitous, sensor-driven environment such as is found in Sentient Computing. Such a model is discussed in Chapter 4.

Chapter 4 first presents an analysis of the deficiencies of existing event models that are based on finite state-machines, when applied to sensor-driven, context-aware systems. It concludes that these impair the expressiveness of the user requirements language. Instead, it proposes a state representation of the knowledge about the context-aware environment in temporal first-order logic (TFOL). Instead of traditional events, our model uses a generalised notion of an event, an *abstract event*, which we define as a notification of transparent changes in distributed state. Abstract Events are discussed in detail in Chapter 8. Referring to the Event-Condition-Action model discussed earlier in this section, a higher-order service *(Abstract Event Detection Service)* accepts a subscription through the *Subscribe()* interface containing an abstract event definition (C) as an argument and, in return, publishes an interface to a further service, an abstract event detector. Abstract Event detectors are structured as Deductive Knowledge Base components and are responsible for determining whether the knowledge defined by the user has been acquired on not. Upon successful detection, an abstract event (E) is published through the *Publish()* interface that notifies the Context-Aware Application Service that the defined system response (A) can now be triggered.

Abstract Event detection is undertaken by the Deductive Knowledge Base component and is discussed in detail in Chapters 8 and 12. Abstract Event Detectors are implemented as Rete networks [38],

and they consist of nodes and arcs. Every time a sensor creates a primitive event, this is translated into a *token*, which is propagated through the arcs to the nodes. Each node checks whether the received tokens correspond to a particular condition, e.g., if they are of class $H\_UserInLocation$. It then forwards the tokens that satisfy the check on to the the child nodes. When a token is forwarded to the final node, an instance of the abstract predicate that is being defined is created or deleted accordingly, and an "activation" or "de-activation" abstract event is triggered, respectively. Rete networks, as used in this thesis, can perform reasoning that is equivalent to TFOL.

Chapter 6 focuses on scalability, as in order to deduce the abstract predicates of interest, the Deductive KB component needs to process a vast number of sensor updates per second (*SensorUpdate()* interface). The user can also provide an AESL definition manually through the *UserUpdate()/UserQuery()* interfaces. *Scalability* is achieved by maintaining a *dual-layer* knowledge representation mechanism for reasoning about the Sentient Environment that functions in a similar way to a two-level cache. The lower layer maintains knowledge about the current state of the Sentient Environment at sensor level by continually processing a high rate of events produced by environmental sensors, e.g., it knows of the position of a user in space, in terms of his coordinates *x,y,z*. The higher layer maintains easily retrievable, user-defined abstract knowledge about current and historical states of the Sentient Environment along with temporal properties such as the time of occurrence and their duration, e.g., it knows of the room a user is in and for how long he has been there. Such abstract knowledge has the property that it is updated much less frequently than knowledge in the lower layer, namely only when certain threshold-events happen. Knowledge is retrieved mainly by accessing the higher layer, which entails a significantly lower computational cost than accessing the lower layer, thus ensuring that the lower-level can be replicated for distribution reasons, maintaining the overall system scalability.

The framework described in this dissertation also supports probabilistic reasoning, not only for inferring richer context such as user movements but also for estimating the likelihood that concrete or abstract knowledge will be acquired in the future. This is undertaken by the Statistical Inferencing Service and it is based on the Naïve Bayes Classifier discussed in Chapter 4. Through the *Fact()* interface the Statistical Inference Service receives both updates on concrete knowledge (sensor updates) as well as abstract events from the Deductive KB component's *Publish()* interface and stores these persistently. Unlike the movement updates, which are asserted into the knowledge base at the same rate as the location updates, likelihood estimation facts are created on demand, through the AESL definitions. The service subscribes to AESL definitions that contain likelihood function predicates (Chapter 4). The latter are compiled into Bayesian Discriminant analysis processes that estimate the likelihood of the predicate of interest, with the parameters given in the statement. The deduced facts are treated as additional, abstract context sources and are published using the *FactLikelihoodUpdate()* interface to the appropriate knowledge base components, where they are included in the reasoning.

1.5 Contribution of the Thesis

The thesis of this dissertation is that it is both feasible and beneficial to provide a framework for context-awareness in sensor-driven systems. In particular, it emphasises the following:

- Existing distributed systems models are unsustainable for sensor-driven, context-aware systems. The framework that is described in this dissertation constitutes an alternative solution for modelling context-awareness in sensor-driven systems.

- This framework is most efficient when it is oriented towards the user and the user's applications; this means that context-aware application development is undertaken by the user, and therefore, it needs to be feasible without undue programming effort. Applications need to be created and removed without stopping or recompiling the system. Whenever appropriate, applications need to remain effective as the user moves from one domain to another.

- For each distributed sensor-driven component, a dynamic model needs to be maintained which integrates knowledge about the entities in the sensor-driven component and the surrounding environment. A language for reasoning about the model is necessary. The language needs to be powerful enough to express human intuition about the state of the physical world, in a way that the instrumenting and processing technology as well as the semantic differences in each distributed model component are *invisible* to the user. Powerful tools for extending the model with abstract knowledge are necessary.
- The framework described in this dissertation, which uses both computer science and engineering methodologies, is a plausible and useful approach to modelling context-awareness and context-aware application development in sensor-driven systems.

In the process of arguing the above thesis, the work described in this dissertation has resulted to the following artifacts:

- A framework called SCAFOS, which bears the following features:
  - It implements a state-based, formal model for context-awareness in sensor-driven systems, that integrates knowledge while maintaining knowledge integrity. For each sensor-driven component, a separate model is maintained. A *state-based* representation is a novel way of modelling distributed systems and it is used instead of traditional-event based models, that are insufficient for sensor driven systems.
  - Knowledge in each model is maintained in a dual-layer knowledge base. The lower layer maintains concrete knowledge predicates, e.g., it knows of the position of a user in space, in terms of his coordinates *x,y,z*. The higher layer maintains abstract knowledge predicates about current and historical states of the Sentient Environment along with temporal properties such as the time of occurrence and their duration, e.g., it knows of the room a user is in and for how long he has been there. Abstract predicates change much less frequently than concrete predicates, namely only when certain threshold events happen. Knowledge is retrieved mainly by accessing the higher layer, which entails a significantly lower computational cost than accessing the lower layer. In this way, this scheme acts as a dual-layer cache for knowledge predicates.
  - SCAFOS introduces the concept of *abstract events*. An abstract event is a novel concept of a generalised event defined as a change in the value (or a reminder) of an abstract state predicate. SCAFOS introduces the AESL language for defining new abstract events from concrete and abstract knowledge predicates. AESL is a TFOL-based language. Phrases in this language can capture human intuition about system state that was not definable before. Such phrases involve negation (e.g., *empty* room) and semantically transparent reasoning with global state (e.g., locate the closest empty meeting room) even when this refers to more than one model with different entities. Furthermore, the AESL language is designed with implementation efficiency in mind. AESL phrases are compiled into reasoning structures called abstract event detectors, which are optimised for computational efficiency. Abstract event detectors are implemented as Rete networks. AESL is complemented by the filtering language AEFSL. The combined use of AESL and AEFSL avoids duplication of computation.
  - SCAFOS contains the SCALA language for using the SCAFOS framework. SCALA is an XML wrapping of the following three sublanguages: The AESL language, the AEFSL language and the ECAAS language. The ECAAS language is an extended ECA language for building applications easily, by binding AESL definitions to action predicates that represent actions available in the environment. It also contains *else* statements and statements for controlling the life-cycle of the created application.

  - SCAFOS contains powerful tools for extending the model with abstract knowledge by means of probabilistic statistical inference, using *Hidden Markov Models (HMMs)* and *Bayesian prediction*. More specifically, SCAFOS contains an HMM-based scheme for detecting and recognising human movements from position streams. This system is independent of user and domain. SCAFOS also contains a scheme for estimating the likelihood of future concrete or abstract predicates holding. This has a number of benefits such as that it enables decision making in the absence of knowledge sources.
  - SCAFOS is dynamically extensible. *Dynamic extensibility* refers to the ability to modify the modelled physical entities and create or remove applications without taking the sensor-driven system offline and without having to recompile existing applications.
  - SCAFOS contains tools for checking the correctness of user requirements and their compatibility with the models of the distributed components.

- An implementation of SCAFOS, based on CORBA [21], consisting of the following services:
  - A Context-Aware Application Service, which enables the simple development of context-aware applications with little programming overhead.
  - A Statistical Inference Service, which detects human movement from location data and estimates the likelihood that an instance of either concrete or abstract knowledge will hold in the future.
  - An Abstract Event Deduction Service, which publishes state changes into abstract events.
  - A set of Deductive Knowledge Base (KB) components that detect abstract events that are published by the Abstract Event Detection Service. Deductive KB components are capable of scalable abstract reasoning.
  - A Satisfiability Service that proves the correctness of user requirements and their semantic compatibility with the sensor-driven system components of the distributed infrastructure.

1.6 Relation to Database and Enterprise Resource Planning Systems

This thesis takes the approach that the design of context-aware applications can be facilitated and automated if it is built on a framework that integrates knowledge about context-aware environments. One result of this is that application development is equivalent to defining a high-level specification and letting the system compile this into a process that creates a new application from the user specification.

This approach in unifying knowledge in context-aware systems is inspired by a similar unification in the modelling of enterprise data that led to the development of *relational databases* and *Enterprise Resource Planning (ERP)* systems. These systems replaced of previous ad hoc data management efforts that suffered from data redundancy and replication of computation. A brief background on these systems follows.

A *relational database* integrates the data used in an organisation under centralised control. Applications sharing the integrated data do not need to know how the data is organised and how it can be accessed. Knowledge is stored once, thus avoiding redundancy and the integrity of such knowledge is maintained. *ERP* systems aim to integrate not only data but also all functions across a company into a single system that can serve all the different departments' particular needs. The ERP system is an integrated software program that runs off a single database so that the various departments can more easily share information and communicate with each other. For example, the processing of a customer order causes the creation of a new process that is executed by a pipeline of system components. The change introduced by the processing of the new order will be propagated to the warehouse management system which will subtract the sold item from the stock, to the company's material planning system which will

calculate new estimates in purchasing raw materials that are caused by the sale. While executing this process, the system maintains integrity constraints between the components.

The most important benefits introduced by relational databases and ERP systems are *reduction of redundancy*, *avoidance of inconsistency, standardisation* and *data independence*.

The methodology by which software for ERP systems is developed has been followed during the implementation described in this dissertation. ERP systems, such as SAP [105], offer middleware support for building the components of the ERP system and integrating legacy systems to the integrated knowledge. Chapter 12 presents similar language-based middleware support that supports maximised, scalable integration. Similarly to the model presented in this dissertation, ERP systems reflect an interdisciplinary development effort. They consist of conceptual building blocks that relate to Business Management Practice, Information Technology and the Specific Business Objectives of the corporation. Similarly, the model presented in this dissertation uses computer science ideas and engineering principles to model and improve context-awareness. The specific context-aware services are independent of the nature of the modelled environments, whether these are academic, corporate or, ideally, mixed.

1.7 Research Limitations

Significant to this research, but not addressed directly in this dissertation, are *security* and *systems networking* research, routing protocols such as multicasting, peer-to-peer routing etc. Mobility in terms of portable code as in *mobile agents* is also complementary, and it is the subject of parallel research. *Grid Computing* is complementary to this thesis in terms of scalability in computational resources and can be seen from a context-aware point of view. *Sensor networks* is an emerging paradigm of sensor-driven systems, where each sensor or sensor cluster forms a network node. Sensor networks are by their nature distributed, and the research described in this thesis is highly applicable to sensor networks as well.

1.8 Logic

Temporal first-order logic (TFOL) is used in this dissertation as a tool for modelling and reasoning about knowledge in sensor-driven systems. Chapter 5 discusses a model for defining knowledge using TFOL predicates in sensor-driven systems. Chapters 6 (Knowledge Representation and Scalable Abstract Reasoning), 7 (Query Analysis and Optimisation), 8 (An Extended Publish/Subscribe Protocol Using Abstract Events) and 12 (Sentient Computing Applications Language: SCALA) all look at the model from different perspectives. Chapter 6 focuses on scalable representation and reasoning with knowledge through *queries*. Chapter 7 discusses an optimisation process for the queries of Chapter 6. Chapter 8 looks at extending the model with new abstract predicates, in an event-based asynchronous interaction. Following Chapter 5, arguing that existing event-based models are not adequate for sensor-driven systems, a new concept, that of an *abstract event*, is introduced in Chapter 8. New abstract predicates are defined by applications through a set of *rules*. A language for creating applications that operate in the model is presented in Chapter 12.

There are two types of TFOL formulae in this dissertation, *queries* and *rules*. Rules are added to the system through the *Subscribe()* interface in Figure 1.1. Queries are added by the user through the *UserQuery()* interface. Queries are discussed in Chapters 6 and 7 and Rules in Chapters 8 and 12.

Rules are TFOL wffs (well-formed-formulae) that resemble Horn Clauses in that there is a single conclusion and a single implication. Rules contain negation, existential quantification and universal quantification, which is a significant part of the contribution of this thesis, as existing event models, such as event composition, have limitations in implementing these operators in a way that is appropriate for the application logic in sensor-driven systems. The right-hand side (RHS) of each rule is an atomic formula defining the abstract predicate that is created by the rule. Queries are syntactically identical to the left-hand-side (LHS) of a rule and they return a set of selected values of the predicate instances to

which they apply, or the answer "no" if the instance defined by the query does not exist. In both queries and rules, universal quantification is implicit. As a convention, constants in TFOL formulae have the first letter capitalised and variables are in lower-case.

Both queries and rules are implemented by Rete networks. Rete networks that implement queries are discussed in Chapters 6 and 7. Rete Networks that implement rules are referred to as abstract event detectors, whenever such rules are used in order to define abstract events (Chapter 8 and 12). The efficiency of different architectures of Rete networks is discussed in Chapter 7. The findings of that chapter apply, therefore, to both queries and rules.

The temporal notation used in this dissertation is discussed in Section 2.7 where some background on temporal logic is given, as well. Belief is also modelled in the TFOL framework, by extended predicates that include uncertainty in their semantics, e.g., *Probability(args), Likelihood(args)* and with *confidence levels*, e.g.,

$$Prob(85\%, 80\%, H\_UserInLocation(uid, \textit{Supervisor}, rid, \textit{Supervisor's-office}), Today)$$

where 85% is the value of the probability estimation and 80% the confidence level that reflects the success rate of the estimation. Belief is modelled as above for simplicity and ease of reasoning. The alternative would be a logic that supports partial truth such as *fuzzy logic* [109]. The *decision making* is not part of the logic. The user is presented with knowledge predicates that are perceived or inferred by the Statistical Inferencing Service of Figure 1.1 associated with a confidence level that reflects the inherent uncertainty in that predicate. The user can make decisions based on such predicate instances.

1.9 Nomenclature

This section gives some definitions of the most common terms described in this dissertation and it discusses the adopted nomenclature. More detailed definitions are included in Appendix B.

A large part of this dissertation (Chapter 7 and Appendix B) investigates finite state machines. These are often referred to as *FSMs*. A *knowledge base* K is a system that stores knowledge about the context-aware environment. A knowledge base represents *predicates* that are *true* by storing an instance of each of these predicates. We refer to this instance as a *fact*. The *assertion* of a *fact* in the knowledge base is equivalent to it being *stored* in the knowledge base as a *true* statement. A fact being *retracted* from the knowledge base results in the *removal* of the fact from the knowledge base. In fact, the *assert* command is similar to a database ADD, whereas the *retract* command is equivalent to a database DELETE. When a fact is *asserted* in the knowledge base, this signifies that the predicate that the fact's predicate has the value TRUE. When the fact is *retracted* from the knowledge base, this signifies that the corresponding predicate has the value FALSE. This nomenclature is taken from logic programming.

For the description of the predicates, wherever these are used in the context of the implementation, a *named parameter notation* is used, which is based on the CLIPS [19, 20] syntax. Table 1.1 uses positional parameter notation for the description of the predicates e.g., $L\_InRegion(X, Y, Z, Rid, Rattr)$ as opposed to the CLIPS notation e.g, $(L\_InRegion(x\ X)(y\ Y)(z\ Z)(rid\ Rid)(rattr\ Rattr))$. Predicates are also often referred to by an appropriate abbreviation. Furthermore, an appropriate prefix is used to differentiate low-level (L_) from high-level (H_) predicates. Low-level (concrete) predicates and high-level (abstract predicates) are discussed in detail in Chapter 6.

The predicate $L\_UserAtPosition(uid, role, x, y, z)$ represents the position of user with identification uid and role $role$, in terms of the coordinates x, y, z. The predicate $H\_UserInLocation(uid, role, rid, rattr, timestamp)$ represents the region with identifier rid and property $rattr$, that contains the user uid with role $role$ at the point of *local* time denoted with timestamp $timestamp$. The predicate $L\_AtomicLocation(rid, rattr, polygon)$ represents the region with identifier rid and property $rattr$ that is bound by the polygon $polygon$, in a given coordinate system. The predi-

Predicate	Abbreviation
$L\_UserAtPosition(uid, role, x, y, z)$	*UP*
$H\_UserInLocation(uid, role, rid, rattr, timestamp)$	*UL*
$H\_UserColocation(uid\text{-}list, role\text{-}list, rid, rattr, timestamp)$	*UC*
$L\_AtomicLocation(rid, rattr, polygon)$	*AL*
$L\_InRegion(x, y, z, rid, rattr)$	*IR*
$H\_EmptyLocation(rid, rattr, timestamp)$	*EL*
$H\_ClosestLocation(uid, role, rid, rattr, timestamp)$	*CL*
$H\_ClosestEmptyLocation(uid, role, rid, rattr, timestamp)$	*CEL*
$H\_ClosestNonEmptyLocation(uid, role, rid, rattr, timestamp)$	*CNEL*

Table 1.1. Most common predicates and their abbreviation.

cate $H\_UserColocation(uid\text{-}list, role\text{-}list, rid, rattr, timestamp)$ represents the fact that a list of users ($uid\text{-}list$) are co-located inside a region with identifier rid and property $rattr$ at a given point in *local* time denoted by the timestamp $timestamp$. The predicate $L\_InRegion(x, y, z, rid, rattr)$ represents the fact that a position (x, y, z) is contained within a region with identifier rid and attribute $rattr$. The last four predicates represent an empty location, the closest location and the closest non-empty location to a user. Note that when constants are assigned to predicate attributes, the first letter is capitalised. For example, Room 10 is modelled by the predicate instance *L_AtomicLocation(Room10, Office,Polygon10)*.

A *model* in this dissertation (except when used in the context of HMM and Bayesian inferencing schemes, where it refers to the respective techniques) is a logical interpretation [65] that satisfies the set of TFOL predicates presented in Chapter 4. The word *model* is also used in this dissertation in connection with the terms *state-based* vs. *event-based* modelling to mean providing a representation of the behaviour of a distributed system using the formalism of *state* or *event* respectively. The term *framework* is used in this dissertation to mean an implementation of a model and a set of tools used for a particular goal, i.e., a language for application development. SCAFOS, the framework proposed in this dissertation, is summarised in Section 1.5 and discussed in more detail in Chapter 10. The name of the proposed framework, "SCAFOS", is a Greek word for "vessel", and it is appropriate here as it represents the structure that contains the tools for creating context-aware applications. The name of the proposed language, "SCALA", has the meaning of *escalation* in Greek. This is also appropriate, as SCALA is used in order to synthesise rich context from concrete context. A lot of examples are implemented in the Laboratory for Communication Engineering of the University of Cambridge, abbreviated as LCE.

1.10 An Interdisciplinary Approach

The approach of this thesis has been to investigate the applicability of the findings of related research areas in computer science and engineering in modelling context-awareness in Sentient Computing. Elements of computer science disciplines such as *automated deduction*, *programming language design* and *distributed systems*, as well as engineering disciplines such as *statistical probabilistic reasoning* and *software engineering* have been essential and successful in supporting the thesis of this dissertation. The most important modelling techniques presented in this dissertation are summarised next.

1.11 Hidden Markov Models

A Hidden Markov Model (HMM) [32, 88] is a stochastic model with an underlying stochastic process that is not observable (it is hidden) but it can only be observable through another set of stochastic processes that produce the sequence of observations.

Processes inferred from position data, such as human movements, are good candidates for HMM-based modelling because they have a number of interesting properties. As users are free to move into space, a position is only dependent on a previous position. HMMs are appropriate for modelling processes with inherent temporality that unfold in time, where a state at time t is influenced only by the state at *t-1*. The constraints of physical space and human movement introduce invariances that can be used as the underlying hidden stochastic process. HMMs are particularly efficient for models that are based on data that is missing or incomplete. Location data belongs to this category. Location samples are often of variable length as the sampling frequency of the location system varies, as the user moves around at different speeds and as some of the sampled sightings are discarded, due to errors during a statistical filtering process, internal to the location system. For all the above reasons, HMMs are considered appropriate tools for modelling inferred knowledge such as human movements in Sentient Computing.

1.12 The Naïve Bayes Classifier

The *naïve Bayes classifier* is a widely used practical learning method, similar in efficiency to neural networks. It applies to problems of *classification* where each instance x is described by a conjunction of attribute values where the target function $f(x)$ is unknown but it can take any value from a finite set V. A set of training examples of the target function is provided, and a new instance is presented, described by the tuple of attribute values $\langle a_1, a_2 \cdots a_n \rangle$. The Bayes classifier is used in order to predict the target value for the new instance. The target value for this instance is assigned the most probable conditional probability value u_{MAP}, given the attribute values $\langle a_1, a_2 \cdots a_n \rangle$ that describe the instance.

$$U_{MAP} = \arg\max_{u_j \in V} P(u_j | a_1, a_2 \cdots a_n) \tag{1.1}$$

This thesis uses the above classifier in order to estimate the likelihood that a predicate instance will occur. For example, considering the predicate

$$H\_UserInLocation(uid, rid, role, rattr, timetamp),$$

the naïve Bayes classifier can be used in order to classify any of the attributes uid,rid, $role$,$rattr$,$timestamp$ to the most probable value of another attribute, given that the values, of some or all of the other attributes, are given. For example, the uid can be classified into the region with the highest probability of being associated with a given uid, based on historic location data, at a given time. This is equivalent to estimating the most probable location that a user with identifier uid will be "seen" by the location system at a given time. Similarly, the predicate[2]

$$H\_ClosestEmptyLocation(John, Sysadmin, rid, Kitchen, 9am),$$

can be used to classify the most probable closest, empty room of type $rattr = Kitchen$ in respect to user with $uid = John$ and property $role = Sysadmin$. This is equivalent to estimating the likelihood that John will be seen in any kitchen at 9 am.

1.13 SCALA vs. SQL

Because of its strong links with the deductive knowledge base component in SCAFOS, SCALA bears strong similarities to languages such as SQL that are used in relational databases for Data Manipulation (DML) and Data Definition (DDL). SCALA also contains support for Data Definition and two Data

[2] As a convention constants such as "John" have the first letter capitalised while variables such as "rid" are in low-case.

SCALA	**SQL**
$CL(uid, role, rid, rattr)$ $\wedge EL(rid, rattr)$ $\Rightarrow CEL(uid, role, rid, rattr)$	CREATE VIEW CEL(uid,role,rid,rattr) AS SELECT EL.rid, EL.rattr, CL.uid, CL.role FROM EL, CL WHERE CL.rid=EL.rid AND CL.rattr=EL.rattr;
rattr=Meeting Room	SELECT * FROM CEL WHERE rattr=Meeting Room;
$< AbsPredDef > EL$ $< parameter >$ *rid string* $< /parameter >$ $< parameter >$ *rattr string* $< /parameter >$ $< /AbsPredDef >$	CREATE TABLE EL rid (CHAR 20) NOT NULL, rattr (CHAR 20);

Table 1.2. SCALA SQL analogies

Manipulation languages AESL and AEFSL. The core entity of the data structure in SCALA is the state predicate, which can be compared to a *view* in a relational database. Predicate instances can be seen as rows in the table that represents the predicate. Concrete predicates can be regarded as tables. Table 1.2 relates SCALA statements to SQL statements.

SCALA and SQL also have a number of key differences: SCALA is far more expressive than SQL in terms of data manipulation, as it can reason with data using FOL operators such as existential quantifiers and negation, thus creating arbitrarily complex views. SQL on the other hand is restricted to operations on sets, in order to create views. Although the SQL CREATE VIEW statement does not contain a definition of the types of the predicates of the view, as those have been already defined in the tables from which they are selected, through a CREATE TABLE statement, in SCALA, the type of the abstract predicate of interest must be defined explicitly. For this reason, Table 1.2 relates type definitions to SQL CREATE TABLE statements. Finally, SQL does not support temporal reasoning. Type definitions are discussed in Section 12.3.

1.14 Thesis Overview

Chapter 2 surveys the most important related literature.

Chapter 3 surveys literature related to existing knowledge inference techniques and demonstrates a movement recognition system that uses HMMs.

Chapter 4 discusses likelihood estimation using the Naïve Bayes Classifier.

Chapter 5 discusses the deficiencies of existing event-based models based on finite-state machines and presents an alternative, state-based conceptual framework in temporal first-order logic.

Chapter 6 presents a scheme for scalable and abstract reasoning for the model presented in Chapter 5.

Chapter 7 discusses query optimisation for the queries of Chapter 6.

Chapter 8 introduces the need for abstract events as notifications of changes in the generated abstract knowledge, and focuses on an extension of the traditional publish/subscribe protocol for abstract events.

Chapter 9 discusses a service that checks the correctness of abstract-event definitions and their compatibility with the integrated knowledge models of distributed, sensor-driven components.

Chapter 10 discusses the implementation of the SCAFOS framework, which provides a complete infrastructure for the dynamic creation and deployment of context aware applications in heterogeneous, distributed sensor-driven components.

Chapter 11 discusses an evaluation of SCAFOS using four example applications.

Chapter 12 discusses the SCALA language for creating, deploying and using SCAFOS in order to create context-aware applications with little programming overhead.

Chapter 13 concludes and discusses interesting extensions to SCAFOS. More specific further work is discussed in the end of each chapter, wherever this is appropriate.

Appendix A contains additional definitions of the model entities of Chapter 5 and illustrates the deficiencies of FSMs discussed in Chapter 5 with examples.

Appendix B summarises SCALA support in terms of Application Programmable Interfaces (APIs) for constructing SCAFOS.

Chapter 2

Background and Related Work

SCAFOS must naturally leverage off the work that preceded it. The purpose of this chapter is to describe previous work in the field of context-aware computing. This chapter looks at context-aware application development from the perspective of how well, if at all, it satisfies the requirements discussed in Section 1.1. Because of the interdisciplinary nature of this dissertation, this chapter also discusses existing work in other areas of computer science and engineering that are relevant to this dissertation, such as automated deduction and statistical inferencing. Although this chapter contains the main volume of the related work, more specific literature is also surveyed inside the relevant chapters. In Chapter 3, after presenting statistical inferencing, existing inferencing techniques that appear in the literature are discussed. Chapter 4, after having introduced the requirements of context-aware applications in sensor-driven systems, surveys literature related to existing event-based models that are based on finite-state machines. Chapter 8, after having introduced abstract events, discusses in detail existing work in the area of asynchronous, event-based communication.

2.1 Location Technologies

Location has been one of the most important and widely used sources of context in context-aware systems. In Chapter 3, location data is used in order to infer user movements and estimate the likelihood that abstract knowledge will be acquired. Location data differ greatly from one implementation to another, and their properties depend on the technology that is used to produce the data. Such technologies employ infrared, radio, ultrasonic or optical sensing techniques. They provide information about the proximity of an object to a sensor, or they give coordinates of the object to some degree of precision. Some also provide information about the orientation of the object that is to be located. Some of the systems can be operated indoors only, outdoors only and some in both. Some key examples are summarised next.

PARCTAB. The Xerox Palo Alto Research Center's Ubiquitous Research program is one of the seminal projects in context-aware computing [97]. The PARCTAB is a personal digital assistant (PDA) that communicates via infrared (IR) data-packets to a network of IR transceivers. The infrared network is designed for in-building use where each room becomes a communication cell. In contrast to the approach used by other PDAs, most PARCTAB applications run on remote hosts and therefore depend on reliable communication through the IR network. The infrastructure provides reliability as well as uninterrupted service when a PARCTAB moves from cell to cell. Each PARCTAB has a pressure sensitive screen on top of the display, three buttons underneath the natural finger positions and the ability to sense its position within a building.

The Active Badge. The Active Badge system [106] was the first indoor badge sensing system; It was developed at Olivetti Research Laboratory. It consists of a cellular proximity system that uses infrared

technology. Each locatable person wears a badge which emits a globally unique identifier every 10 seconds or on demand. A central server collects the data from fixed infrared sensors in the building, aggregates it, and provides an application programmable interface for using the data.

The Active BAT. The Active BAT [41] uses an ultrasound time-of-flight trilateration[1] technique to provide more accurate physical positioning than the Active Badge. Users and objects carry Active BAT tags. In response to a request that the controller sends via short-range radio, a BAT emits an ultrasonic pulse to a grid of ceiling-mounted receivers. At the same time the controller sends the radio frequency request packet, it also sends a synchronised reset signal to the ceiling sensors using a wired serial network. Each ceiling sensor measures the time interval from reset to ultrasonic pulse arrival and computes its distance from the BAT. The local controller then forwards the distance measurements to a central controller which performs the trilateration computation. Statistical pruning eliminates erroneous sensor measurements caused by a ceiling sensor hearing a reflected ultrasound pulse instead of one that travelled along the direct path from the BAT to the sensor. The system can locate BATs to within 3 cm of their true position for 95 percent of the measurements. It can also compute orientation information.

Cricket. The Cricket System [86] is also based on ultrasound technology. In contrast to the BAT, it uses ultrasound emitters to create the infrastructure and embeds receivers in the object being located. This approach forces the mobile objects to perform all their own triangulation[2] computations. Cricket uses the radio frequency signal not only for synchronisation of the time measurement but also to delineate the time region during which the receiver should consider the sounds it receives. Like the Active BAT, Cricket uses ultrasonic time-of-flight data and a radio frequency control signal but this system does not require a grid of ceiling sensors. Cricket in its currently implemented form is less precise than the Active BAT in that it can delineate 4x4 square foot regions while the Active BAT is accurate to 9 cm. However, the fundamental limit of range estimation accuracy used in Cricket should be the same as the Active BAT. Its advantages include privacy and decentralised scalability while its disadvantages include a lack of centralised management or monitoring and the computational burden that processing both the ultrasound pulses and RF data places on the mobile receivers.

RADAR. RADAR [9] is Microsoft's building-wide tracking system based on the IEEE 802.11 WaveLAN wireless networking technology. RADAR measures at the base station the strength and signal-to-noise ratio properties of signals that wireless devices send, then it uses this data to compute the 2D positions within a building. Microsoft has developed two RADAR implementations: one using scene analysis and the other using triangulation. RADAR requires only a few base stations and it uses the same infrastructure that provides the building's general purpose wireless networking; however, it is limited to two dimensions and it can offer a resolution of 2 to 3 meters, which is too low for the needs of most context-aware applications.

TRIP. Several groups have explored using computer vision technology to determine object locations. TRIP [59] (Target Recognition using Image Processing) is a vision-based sensor system that uses a combination of 2-D circular barcode tags or *ringcodes* and inexpensive CCD cameras in order to identify and locate tagged objects in the camera's field of view. Optimised image processing and computer vision algorithms are applied to obtain in real-time the identifier and pose of the moving target with respect to the viewing camera. The advantages of TRIP are that it is low cost and easily deployable.

[1] Trilateration is a method of surveying analogous to triangulation, in which each triangle is determined by the measurement of all three sides.

[2] Triangulation is defined as the measurement of a series or network of triangles in order to survey and map out a territory or region, by measuring the angles and one side of each triangle.

The disadvantages are that the location of the viewing camera needs to be known, and the processing overhead for analysing and optimising the image. Tags can be identified within a 3 m distance from the camera. This renders TRIP appropriate for specific applications that use tags to control the application flow e.g. controlling a power-point presentation with the tags.

GPS. The Global Positioning System (GPS) [26] is a satellite-based navigation system developed and operated by the US Department of Defence. GPS permits land, sea and airborne users to determine their three-dimensional position, velocity and time. GPS uses a constellation of 21 operational NAVSTAR satellites and 3 active spares. The GPS satellite signal contains information used to identify the satellite and provide position, timing, ranging data, satellite status and the updated ephemeris (orbital parameters). A minimum of 4 satellites allows the GPS client to compute latitude, longitude, altitude (with reference to mean sea level) and GPS system time, through a process of triangulation. The satellites receive periodic updates with accurate information on their exact orbits. Differential GPS (DGPS) is regular GPS with an additional correction signal added. DGPS uses a reference station at a known point (also called a 'base station') to calculate and correct bias errors. The reference station computes corrections for each satellite signal and broadcasts these corrections to the remote, or field, GPS receiver. The remote receiver then applies the corrections to each satellite used to compute its fix.

Ultra Wideband. Ultra Wideband (UWB) [34] is a radio technology that opens up new capabilities in radio communications. A wireless technology transmits digital data at very high rates over a wide spectrum of frequency. Within the power limits allowed under current FCC regulations, not only can UWB carry huge amounts of data over a short distance at very low power, but it also has the ability to carry signals through doors and other obstacles that reflect signals at more limited bandwidths and higher power. In addition to its uses in wireless communications products and applications, UWB can also be used for very high-resolution radars and precision (sub-centimeter) location and tracking systems.

UWB radiation has unique advantages: transceivers and antennas can be made very small (i.e., coin size), low power and low cost because the electronics can be completely integrated in CMOS without any inductive components. Ultra Wideband signals form a shadow spectrum that can coexist and does not interfere with the sinewave spectrum. The transmitted power is spread over such a large bandwidth that the amount of power in any narrow frequency band is very small. The advantages of spread spectrum are shared, including multipath immunity, tolerance of interference from other radio sources and inherent privacy from eavesdropping (low probability of intercept). Ultra Wideband / non-sinusoidal signals have very good penetrating capabilities, and they support centimetre-level location accuracy without needing extremely accurate clocks to synchronise multiple receivers.

2.2 Sensor-Driven Paradigms

Taking advantage of sensing technologies such as the ones described above, a number of computing paradigms have been developed, all of which aim to provide awareness of the physical world to their applications. Awareness is used to achieve three particular goals: personalisation of their behaviour to their current user, adaptation of their behaviour according to their location, or reaction to the surroundings. All three actions are undertaken by all of the main context-aware programming paradigms which are summarised below, starting from the seminal principle of *Ubiquitous Computing*.

Ubiquitous Computing. Ubiquitous computing [108] was proposed by Weiser in 1993, as a method of enhancing computer use by making many computers available throughout the physical environment,

but making them effectively invisible to the user. This approach can be seen as opposing the tendency of the computing industry that prevailed at the time, namely, to maximise the integration of processing power and software tools into the Personal Computer, while at the same time, reducing its size as much as possible. On the contrary, Ubiquitous computing followed the development trends that resulted in using a three tier architecture (that pushes the computational load onto a powerful server machine, minimising the computational load on the client), and extended these trends even further. Ubiquitous computing adopted the vision that computational power would not be restricted to dedicated pieces of equipment such as laptops and PDAs but would be distributed in objects of everyday life. Furthermore, computers would stop being the focus of user efforts and would serve as tools that provide a service. In other words, computers would be *invisible* to the user.

Pervasive Computing. Pervasive Computing shares the vision of Ubiquitous computing, i.e., that computing technology should be embodied into real world objects like furniture, clothing, crafts, rooms, etc. Furthermore it advocates that those artefacts also become the interface to "invisible" services and mediate between the physical and digital (or virtual) world via natural interaction. It, therefore, extends and complements Ubiquitous computing with the concepts of minimal technical expertise, reliability and more intuitive interaction.

Sentient Computing. Sentient computing [44] is a paradigm that focuses on *sensing* the environment. This is achieved through a *sensor infrastructure* (Figure 6.1), distributed in the environment, which provides information about the spatial properties of users and objects, i.e., their position in space, their containment within a region such as a room or their proximity to a known physical location. *Sentient Applications* make it possible for the user to easily perform complex computations involving spatial and temporal notions of a dynamic, changing environment. For example, when a user walks into his study at home, it is possible for his PC to automatically and seamlessly display the desktop from his office environment. Or, whenever two people are co-located in a single space, the system can make this information graphically available to an Active Map or deliver a relevant reminder. For example, *"You asked me to remind you to give Tom his book back the next time you meet him."*

Sentient Computing, as is implemented today, abides by the principles of Ubiquitous and Pervasive computing. Although it does not distribute computation to everyday objects, it removes all the burden of computation from the user who interacts with the physical environment through using a small object, such as an Active BAT. The user can click on specific areas in the physical space, e.g., presented as posters in order to invoke an available service, such as, the forwarding of a scanned document to somebody's email account, or requesting a notification before leaving the lab.

An obvious limitation to this interaction scheme is that the user can only select an already existing application, but cannot create additional requirements. This is discussed in more detail in Section 2.4, where SPIRIT [41] is discussed. The realisation of this need inspired the thesis of this dissertation, namely to provide programming support that would allow users to easily create applications for such an environment. Sentient Computing in its current implementations can be made more pervasive by allowing more natural, intuitive interaction; it can also be made more ubiquitous by distributing interaction to physical objects.

Wearable Computing. Wearable Computing [62] is a programming paradigm that uses wearable computers. A wearable computer is a computer that is subsumed into the personal space of the user, controlled by the user, and has both operational and interactional constancy, i.e., is always on and always accessible. Most notably, it is a device that is always with the user, and into which the user can always enter commands and execute a set of such entered commands, even while walking around or doing other activities. The most salient aspect of computers, in general, is their reconfigurability and their generality, e.g., that their function can be made to vary widely depending on the instructions provided for program

execution. With the wearable computer (WearComp) this is no exception, e.g., the wearable computer is more than just a wristwatch or regular eyeglasses: it has the full functionality of a computer system; but, in addition to being a fully featured computer, it is also inextricably intertwined with the wearer. This is what sets the wearable computer apart from other wearable devices such as wristwatches, regular spectacles, wearable radios, etc. Unlike these other wearable devices that are not programmable (reconfigurable), the wearable computer is as reconfigurable as the familiar desktop or mainframe computer.

Several other theories have recently emerged focusing on different aspects of context-aware computing. It is beyond the scope of this dissertation to do more than mention them by name here: Proactive Computing, Calm Computing, Planetary Computing.

2.3 Application Areas

A number of context-aware applications have been designed over the years ranging from augmenting the human to helping manage every day tasks. A taxonomy presented in [96] classifies applications according to the type of system response that is invoked as a result of detecting the appropriate context. These categories are: selecting the most proximate service, automatic reconfiguration, contextual commands or context-triggered actions. A taxonomy that is more focused on the nature of the provided applications and the social context on which is it provided can be found in [89].

This section investigates application development from a human perspective aiming to determine to what extent such applications improve task handling without introducing additional overhead, such as a high rate of untimely interruptions. First, *interruption management* is discussed, and it is concluded that a user-centered model is essential for providing an effective service. Next, the most relevant and interesting applications and the degree to which interruptions can be handled are discussed.

Up to now, there has been little progress in the area of managing interruptions. Dey et al. [30] refer to determining interruptibility as an "enormous unresolved research problem" and express interest in including interruption management in the CyberMinder tool which is intended to deliver context-aware reminders. An outline of the key interruptibility issues still to be solved is presented in [104]. The key issue that prevents us from managing interruptions is the fact that we don't know ***if the interruption will be useful to the user at the instant that it is generated.*** An obvious mistake to make would be to assume that all interruptions are undesirable. Indeed, a study has shown that a high percentage of the interruptions that occur in an office environment are useful rather than disruptive [110]. An illustrating example is included in [104] and demonstrates a second issue, i.e., that ***applying a general rule concerning interruption management at one time is not good enough***. For example, an application withholding all interruptions while a user is in meeting may shield a call that contains vital information on the deal in progress. A third issue concerns ***the way decisions are made as to when is a good time to interrupt.*** For example, a period that a user is absorbed in an important though informal conversation just before or after a meeting takes place might be the worst moment to interrupt. Another study has shown that the nature of the individual task and the experience of the subject can greatly influence the effect that interruptions have on performance [76]. These results advocate strongly the need to associate the user personally with determining when interruptions are desirable and when not. This realisation has inspired the focusing of the work described in this dissertation towards designing a user-centered model for context awareness.

Look Out. The need to determine the status of user attention while designing automated User Interfaces is discussed in [45]. LookOut, an assistant to Microsoft's Outlook mailing program, identifies new messages that are opened and attempts to assist users with reviewing their calendar and with composing appointments. In order to determine when to present the user with an action-options menu without distracting him, e.g., while he is still reading his email, the time between when a message is opened and

user action is taken is measured by the system and a model of attention is constructed from this timing study.

CyberMinder. A context-aware system for supporting reminders has been proposed in [30]. The tool aims to support users in sending and receiving reminders that can be associated with situations involving time, place and more sophisticated pieces of context so that reminders can be delivered at the appropriate time. For example, a reminder is sent to a user to remind him not to forget his umbrella as he approaches his front door while leaving the house in the morning. Cyber Minder delivers a reminder when the associated situation has been realized, and chooses the delivery mechanism/context based on the recipient's current context. However, it does not take into account how interruptible the user is. Another disadvantage is that it does not handle synchronous communication at all, e.g., it cannot route telephone calls over IP, according to the user's context. Lastly, complex situations can be composed from sub-situations by a set of operators and operands, but it does not support a generic language for specifying situations.

comMotion. comMotion is a GPS-based system [64] that uses location and time information in order to deliver location-related information, such as a grocery list, when the user is close to a super-market. When a reminder message is created, a location is associated with it. Then, when the recipient arrives at the specified location, the messages associated with that location are delivered via speech synthesis. As with CyberMinder, comMotion does not handle synchronous communication and does not take the recipient's interruptibility into account.

Nomadic Radio. Nomadic Radio [95] is a system developed by MIT that aims to handle asynchronous message delivery based on message priority, content, usage level and environmental noise. The system attempts to tackle interruptibility issues by adapting the way it notifies the recipient based on the level of the ambient noise in the environment. For example, if the conversation level in a room is high, the auditory notification is discreet and the user can choose to suppress the summary which will follow. Interruption filtering is done based on priorities, and high priority calls are put through no matter what context the recipient is in. However, Nomadic Radio does not handle any richer context than noise level and only has 2 modes. *Busy* and *non busy*. For these modes it only makes exceptions for high priority calls. In fact it pays no heed to the content of each interruption.

Proem. Proem [54] is a wearable computer-based system that supports profile based cooperation. Wearers can write simple rules that indicate their interests in other people. When someone physically close to the wearer has a profile that matches one or more of his interests, Proem can alert him. Interests are limited to names, personal interests and hobbies. Although Proem uses user profiles to define the personal details of a user, it only contains limited fields and a restricted grammar. As far as context is concerned, it only takes into account co-location between 2 users.

Memory Glasses. Memory Glasses is a wearable, context-aware, reminder system [28]. It deals with activity/location/time-based reminders, but only handles limited context and has limited options for users to specify their own rules. It also takes no heed of user interruptibility.

Remembrance. In 1996, Rhodes [90] at the MIT Media Lab developed a wearable ***Remembrance agent***. It is a continuously running proactive memory aid that uses the physical context of a wearable computer to provide notes that might be relevant in the user's current context. It displays one-line summaries of note-files, old email, papers and other text information that might be relevant to the user's current context. The summaries are listed in the bottom few-lines of a heads-up display. The Remembrance agent uses five kinds of context: the wearer's physical location, people who are currently around, subject field, date and time-stamp and the body of the note.

Teleporting. The Teleporting System [91], developed at Olivetti Research Laboratory, is a tool for creating mobile X sessions. It provides a familiar, personalised way of making temporary use of X displays as the user moves from place to place.

2.4 Development Platforms

In Chapter 1, a set of features was introduced that are considered essential for modelling the building and execution of context-aware applications. In this section, existing architectures built to support context-awareness are discussed, focusing on the level of support (existing or proposed) for the above mentioned features. It will be noted that none of these architectures provides the required support. This realisation validates the need for the new conceptual framework for application development that is presented in this dissertation.

Stick-e Notes. The Stick-e Notes system [13] is a general framework for supporting a certain class of context-aware applications. More specifically, it aims to support application designers in actually using context to perform context-aware behaviours. It provides a general mechanism for indicating which context an application designer wants to use and provides simple semantics for writing rules that specify what actions to take when a particular combination of context is realised. This approach, while it shares some of the goals of this thesis, appears to be quite limited. The semantics for writing rules is quite limited, and there is no support for high-level knowledge. The only supported actions seem to be document creation. It provides a separation of concerns between the way knowledge is sensed and the way it is used. The knowledge aggregation is centralised without enabling scalability. There does not seem to be any scope for distributed reasoning that is independent of the implementation.

CoolTown. CoolTown [17] is an infrastructure that supports context-aware applications by representing each real world object, including people, places and devices, with a Web page. Each Web page dynamically updates itself as it gathers new information about the entity it represents. This process is similar to aggregating knowledge for that entity. CoolTown is primarily aimed at supporting applications that display context and services to end-users. For example, as a user moves throughout an environment, a list of available services for this environment is presented. Cooltown does not support other context-aware features, such as automatically executing a service based on context. Its knowledge aggregation mechanism is not scalable, and it is not clear how reasoning about knowledge about more than one entity can take place. It is not designed to support storage of context.

The Context Toolkit. The Context Toolkit [31, 94] is the implementation platform of a software architecture for the development of context-aware applications. It supports modular design that consists of context widgets, context interpreters, context aggregators and context services. It also offers a set of libraries with a number of pre-constructed such components.

The Context Toolkit is quite sophisticated in terms of the architectural requirements for supporting the building of context-aware applications and shares some of the design principles of this dissertation. It provides *context widgets* which are abstractions of concrete context that abstract away the details of context acquisition. This implements a separation of concerns between the way context was acquired and the way it can be used. Context widgets have an inbuilt degree of scalability in the spatial properties of entities. I.e., a number of widgets can be used in order to accommodate a large number of context sources. The location context widget can filter through only changes in user locations that signify a change of room, however, there appears to be no support for generalising this feature in different contexts. *Context interpreters* offer some abstraction over spatial properties such as converting room locations into building locations, however, the implementation of the abstractions is up to the developer of these

components and it cannot be done dynamically. *Context aggregators* aggregate context sensed for a single entity from a variety of sources. However, it appears to be infeasible to reason about the state of multiple entities. In fact the aggregators seem to be designed so that they only aggregate context for a single entity. Furthermore, there does not seem to be any checking for whether the accuracy of the source is acceptable to the application. *Context services* are concerned with actions rather than *context acquisition* and offer simple development of actions that are to be triggered as a result of the context aggregator. The communication between the platform components is distributed and event-based. However, it is not transparent as each component publishes its own events. There is also no support for semantic interoperability between heterogeneous components that would allow reasoning transparently about distributed state.

SPIRIT. The SPIRIT project (SPatially Indexed Resource Identification and Tracking)[41, 1] is a sophisticated platform that integrates configuration data for heterogeneous networked hardware and software with fine-grained location data for users and equipment, allowing software to dynamically configure and reconfigure itself as users move around the networked environment. The ultimate goal of the project is to make it seem to users as though an application knows as much about the physical environment as they do. To achieve this goal, information is gathered from a variety of sensor sources, including the Active Badge, the Active BAT and telemetry software monitors (for keyboard, CPU, disk and network traffic).

In order to achieve the above, SPIRIT provides a platform for maintaining spatial context based on raw location information derived from the Active BAT location system. SPIRIT has a similar goal to our proposed architecture in that it offers applications the ability to express relative spatial statements in terms of geometric containment statements. It is inherently scalable both in terms of sensor data acquisition and management as well as software components. Its approach towards both data-modelling and scalability is quite different from the one adopted in this thesis. SPIRIT models the physical world in a bottom-up manner, translating absolute location events for objects into relative location events, associating a set of spaces with a given object and calculating containment and overlap relationships among such spaces, by means of a scalable spatial indexing algorithm. However, this bottom-up approach is not as powerful in expressing contextual situations as the top-down approach presented in this dissertation, as is established in Chapter 4. Although SPIRIT supports parallelism at the level of the storage of world model objects [103], the scalability of the above mentioned *spatial indexing algorithm* is unclear. In fact, calculating relationships between spaces that are not stored on the same computer suggests a high communication overhead between the distributed elements that may affect significantly the response time of the algorithm.

SPIRIT provides support for some inferencing based on location information, such as whether a user is sitting or standing. It also supports adaptability in the sensor update rate, based on the produced inferences, e.g., the monitoring rate is adapted according to the user's speed.

SPIRIT does not support application development with knowledge other than that which pre-exists in the system, which is, for the most part, directly produced by sensors. Writing applications in SPIRIT requires writing code in C++, using event types that already exist in the system.

Active Badge. The Active Badge has been used also for providing mobility to applications. The Teleporting application [91] uses the location information provided by the Active Badge, and the control signal generated when users press the buttons situated on their Active Badges, to relocate a user's X sessions to a new desktop.

QoSDREAM. QoSDREAM [87] is a research platform and framework for the construction and management of context-aware and multimedia applications. QoSDREAM simplifies the task of building

applications by integrating the underlying device technologies and raising the level at which they are handled in applications. More specifically, QoSDREAM was initiated in order to integrate an event-based architecture derived from the Cambridge Event Architecture (CEA) with a multimedia framework called Djinn [23]. The integration enables the construction of applications such as follow-me, multimedia-based communications. Such an application for a hospital environment is described in [66].

The platform is highly configurable, allowing researchers to experiment with different location technologies and algorithms used to handle the information being generated by these technologies [71]. FLAME (Framework for Location-Aware ModElling) is an open source framework for location-aware applications released by the QoSDREAM project. FLAME is fully distributed, providing services to application programs through CORBA interfaces. The framework is written largely in Java, but applications can be written in any CORBA-compatible language.

QoSDREAM/FLAME separates concerns between knowledge acquisition and use. However it does not offer support for performing abstractions of knowledge. Furthermore, there is no separation of concerns between storing dynamic and static data, which makes the model dependent on the specific domain. Its event mechanism offers asynchronous transparent communication, but no interoperability over heterogeneous environments and no reasoning about state. Lastly, it is not dynamically extensible with new context or applications.

LIME. A platform called LIME for programming reactive, location-aware applications using data stored in a shared, distributed tuple space called Linda is presented in [68]. LIME stores knowledge (for the most part location information) about the distributed environment using a state-based temporal representation. It provides interfaces for storing, deleting and reading knowlege out of the shared tuple space, but does not offer any support for abstracting knowledge from other, already available knowledge, nor does it scale in the presence of a large number of updates.

Realms and States. Realms and States [72] is a model for location-awareness that supports logical contexts. The idea of logical location contexts is presented, which provides enhanced privacy and supports specialised notions of distance, and offers a paradigm that unifies location with other types of context. This is developed in the form of a framework consisting of realms, which are collections of spatial states with realm-maps providing mappings between realms.

2.5 Sensor-Driven Systems

A number of significant projects already exist in the area of sensor-driven systems that deal with data produced by sensors in streams, along with a storage and a query model for such data.

Cougar. A database for sensor data along with a scheme where queries over data are expressed in SQL is presented in [12] as part of the *Cougar* project. In that work, streams are represented as persistent, virtual relations and sensor functions are modelled as abstract data types. Queries are seen as SQL SELECT statements over both stored and sensor data. Stored data are modelled as relations and sensor data are modelled as timeseries. Queries in that scheme are enhanced with support for defining how long the queries last and how often they are applied. Queries can aggregate sensor data over a time period and they can correlate data from multiple sensors, however, it is not clear from the paper how distributed query processing is achieved, as it is not implemented in the original version of Cougar. This makes the notion of simultaneity (e.g., that two sensor readings hold simultaneously) unclear. The scheme does not focus on system issues involved with efficiently executing queries. It does not cater for scalability, which is a significant problem with sensor data, nor does it support queries on abstract event types that are deduced rather than provided directly from sensor data.

Fjords. An architectural construct called *Fjords* for creating queries over sensor streams is discussed in [60]. Queries in this scheme consist of a set of selections to be applied via the selection operator, a list of join predicates and an optional aggregation and grouping expression such as average, maximum etc. By linking operators with queues, a combination of both push and pull operations is possible. *Sensor proxies* constitute mediators between sensors and queries. An example is given where the sensor rate is controlled from the Fjords. In this example, only events that have a significance are transmitted, rather than low-level events, leading to a reduction in the communication cost. However, this seems to be done, in an *ad hoc* way, and no programmable support for determining the desired significance that determines the event rate is discussed.

Gator Networks. Rete networks for condition evaluation in active databases were proposed in [40]. In that work, Gator networks are introduced, which are seen as generalisations of Rete networks where nodes that perform conjunction are allowed more than two inputs. Thus, Gator networks can be optimised using randomised state-space search strategies. This approach is compatible with the one presented in this dissertation, although applied to the context of active databases without considering distribution issues.

2.6 Statistical Inferencing

This section discusses existing work that has also used or attempted to use similar techniques for context awareness or related areas, and it demonstrates that the selected methodologies have produced indisputably significant results in related areas while pursuing sufficiently different or complementary goals from the ones pursued in this dissertation.

2.6.1 Hidden Markov Models (HMM)

Hidden Markov Models [32, 88] have been used traditionally with great success in the fields of protein sequence analysis, speech recognition [88] and natural language processing. Here we discuss three related uses.

Learning Significant Locations and Predicting User Movements with GPS. In this work [4], HMMs are used with location data mainly with a goal to predict the user's destination based on statistical data from his movement patterns. The presented system learns from monitoring user tracks which are the most frequent points where the user stops, and classifies these as significant locations. It then uses HMMs to predict a user's destination, aiming to proactively send that user a reminder relevant to his final destination or to arrange a serendipitous meeting.

SmartMoveX on a Graph - An Inexpensive Active Badge Tracker. In this work [55], a low-cost location system is described. Receivers placed in the building's existing offices connected to existing PCs transmit signal strength readings to a central PC using the building's existing computer network. Combined with the low cost of the hardware, using the existing network makes this active badge system much less expensive than many others. To compute locations based on signal strength, signal strength readings from predefined location nodes in the building were gathered. A graph on these nodes was defined which allowed for enforcing constraints on computed movements between nodes (e.g., cannot pass through walls) and to probabilistically enforce expectations on transitions between connected nodes. Modelling the data with a Hidden Markov Model, optimal paths were computed based on signal strengths over the node graph.

Movement Awareness in Sentient Computing. A system that can observe, recognise and analyse human movements in order to provide this awareness to context-aware applications is presented in [42]. The system uses the ground reaction force to classify and analyse movements in a non-clinical environment. The signal is classified using statistical pattern recognition. Equipped with knowledge of the movement, characterisation is achieved by analysing the ground reaction force to extract parameters of the movement.

Although movement characterisation is also addressed in this dissertation, a different approach is adopted. In the work described in [42], ground force is measured by sensors underneath the floor, requiring enhanced infrastructure. Movement awareness in this thesis utilises a location sensing technology of similar accuracy to the Active BAT. Furthermore, the focus of movement characterisation is different, as [42] characterises a single step whereas this thesis characterises walking samples, independent of how many steps they comprise. The recogniser described in [42] does not seem to be tested with different individuals nor is it able to perform user recognition.

2.7 Logic

Using logic to support reasoning about knowledge in sensor-driven systems is one of the fundamental principles of this dissertation. This section presents background work on logic, focuses on *temporal first-order logic (TFOL)* and justifies the assumptions made when TFOL is used as a modelling tool.

First-Order Logic (FOL). First-order predicate calculus or first-order logic (FOL) [65] is a theory in symbolic logic that states quantified statements such as "there exists an object such that..." or "for all objects, it is the case that...". First-order logic is distinguished from higher-order logic in that it does not allow statements such as "for every property, it is the case that..." or "there exists a set of objects such that...". Nevertheless, first-order logic is strong enough to formalise all of set theory and thereby virtually all of mathematics. It is the classical logical theory underlying mathematics. It is a stronger theory than sentential logic, but a weaker theory than arithmetic, set theory, or second-order logic.

Like any logical theory, first-order predicate calculus consists of

- A specification of how to construct syntactically correct statements (the well-formed formulae),
- a set of axioms, each axiom being a well-formed formula itself,
- a set of inference rules which allow one to prove theorems from axioms or earlier proven theorems.

There are two types of axioms: the logical axioms which embody the general truths about proper reasoning involving quantified statements, and the axioms describing the subject matter at hand, for instance axioms describing sets in set theory or axioms describing numbers in arithmetic. While the set of inference rules in first-order calculus is finite, the set of axioms may very well be and often is infinite. However, a general algorithm is required that can decide for a given well-formed formula whether it is an axiom or not. Furthermore, there should be an algorithm that can decide whether a given application of an inference rule is correct or not.

Temporal First-Order Logic (TFOL). Temporal first-order logic is an extension of first-order logic that includes notation for reasoning about temporal durations during which statements are true. There are various sorts of temporal first-order logic. Some common sorts include three prefix operators represented by a circle $\circ$, square $\Box$ and diamond $\Diamond$ which mean "is true at the next time instant", "is true from now on" and "is eventually true". An additional operator, the "Until" operator is often available. The expression xUy means x is true until y is true. However, in this dissertation a different approach is adopted. The method of *temporal arguments* is selected as a temporal notation that is appropriate for the application

Operator	Description
$e_1; e_2$	e_1 before e_2
$e_1; e_2 ! e_3$	e_1 before e_2 without e_3 in between
$\|e_1, e_2\|_{T=t_1}$	e_2 happened within time t_1 from e_1.
now (implicit)	At the current instant.
timestamp	At time t.

Table 2.1. Temporal operators in SCAFOS.

logic in sensor-driven systems. Chapter 6 discusses a scheme for representing and reasoning about two levels of knowledge. Sensor-level predicate instances naturally reflect the current instance, however, there is often a need to reason about past (historic) instances that were active within a certain duration or with other high-level knowledge which is *deduced* from the sensor data. In the adopted notation, the temporal dimension is captured by augmenting each current high-level predicate instance with an extra argument to be filled by a timestamp, according to a local clock, e.g., *H_UserInLocation(John,PhD,Room 10,Office,12:34)*. Current sensor-level predicates do not have timestamps and the concept of *now* is implicit. *Historic* predicate instances are augmented with two timestamps, the first denoting the point in time when they started holding and the second denoting the point in time when they stopped holding. For example,
H_UserInLocationHistoric(John, PhD, Room 10,Office,12:34,12:37).

Temporal Operators in SCAFOS. Apart from atomic predicates, this dissertation uses TFOL for more complex formulae such as *queries* and *rules*. Rules are TFOL wffs (well-formed-formulae) that resemble Horn Clauses in that there is a single conclusion and a single implication. The right-hand-side (RHS) of each rule is the abstract predicate that is created by the rule. Queries correspond to the left-hand-side (LHS) of a rule and they return a set of selected values of the predicate instances to which they apply, or the answer "no" if the instance defined by the query does not exist. In both queries and rules, universal quantification is implicit.

A set of temporal operators has been designed and implemented in this dissertation, which operate on FOL predicates. These operators are more specific than the four general operators of TFOL and they are tailored to the application logic of sensor-driven systems. Their design has been influenced by the work presented in [63, 81]. For example, the term $|e_1, e_2|_{t>300}$ denotes the fact that event e_2 follows e_1 by at least 5 minutes. The operators proposed in this thesis are shown in Table 2.1 and are discussed in more detail in Chapter 12.

Description Logics. Description Logics [73] are logic-based approaches for knowledge-representation systems. Description logics represent entities using a "UML"-like language called DL Language. The logic used for reasoning about such entities is derived from first-order logic but is much less powerful and expressive than first-order logic. Description Logics have the advantage that they *allow* for the specification of *logical constraints*. Any potential contribution of Description Logics to Sentient Computing is yet to be determined.

SPASS. SPASS [107] is a theorem prover for FOL with equality. Given a theorem in FOL and a set of axioms, it finds a proof if the theorem can be proven from the axioms. SPASS is a powerful tool because it has an online interface, whereby theorems can be submitted and tested for satisfiability, as well as tools for translating FOL to Clause Normal Form (CNF) which can then be converted to propositional logic such that it can be used by other satisfiability tools. In this dissertation, SPASS is used in a novel way, namely, as the core of the Satisfiability Service of Figure 1.1. More specifically, a formula in FOL that defines an abstract event (AESL definition) is asserted in the prover, which treats it as a theorem that

needs to be tested for satisfiability given some appropriate axioms. Chapter 7 discusses this in detail.

Fuzzy Logic. Fuzzy Logic [109] is a superset of Boolean logic dealing with the concept of partial truth. Whereas classical logic holds that everything can be expressed in binary terms (0 or 1, black or white, yes or no), fuzzy logic replaces Boolean truth values with degrees of truth which are very similar to probabilities (except that they need not sum to one). This allows for values between 0 and 1, shades of gray and maybe; it allows partial membership in a set. It is highly related to fuzzy sets and probability theory. It was introduced in the 1960s by Dr. Lotfi Zadeh of UC Berkeley.

2.8 Production Systems

A *production system* is a programming language that does not require the programmer to specify how the various parts of the program will interact. A production system interpreter is a computer which contains two separate memories called *production memory* and *working memory*. Production memory holds the expressions (productions) to be executed by the processor and working memory holds the data operated on by the program.

A production system incorporates no concept of sequential control flow through the program. The flow control in a production system is determined by the order in which the condition parts of the productions become true. The interpreter repeatedly executes the following steps:

1. Determine which productions have true condition parts.
2. If there is no production with true condition parts, halt the system. Otherwise, select a production from those that do.
3. Perform the actions specified by the chosen production.
4. Go to step 1.

This sequence is called the *recognise-act* cycle. Step 1 of the cycle is called the *match*. Step 2 is called *conflict-resolution* and step 3 is called *act*.

A *production* is a list that may contain a *condition part* followed by a right pointing arrow. For example, consider a production system that is interested in Greek tragedy. Its working memory would contain the following data elements:

$$Agenor \textit{ is-father-of } Cadmus.$$
$$Cadmus \textit{ is-father-of } Polydorus.$$
$$Polydorus \textit{ is-father-of } Labdacus.$$
$$Labdacus \textit{ is-father-of } Laius.$$

The objects contained in production memory are condition-action pairs called productions: "If there is a man whose father is Laius, then assert that the man killed Laius." This can be symbolised with the following phrase:

$$(Laius \textit{ is-father-of } ?X) \Rightarrow (assert\ (?X \textit{ killer-of } Laius)) \tag{2.1}$$

In order for the production to be triggered, the working memory needs to contain the element: *Laius is-father-of Oedipus*.

The *condition part* of the production is also a list which may contain several condition elements in the form of *patterns*. The match tries to find instances of the class defined by the pattern among the

lists in working memory, a process called *instantiating the pattern*. A production is ready to be executed when all its non-negated condition elements are instantiated and none of the negated ones. The ordered pair of a production name and the collection of data elements that instantiate the production's condition elements is called an *instantiation*. The set of all legal instantiations is called the conflict set.

2.8.1 The Rete Algorithm for the Many Pattern to Many Object Pattern Match Problem

The Rete algorithm [37, 38] is a powerful and efficient algorithm for pattern matching as it exploits properties such as *temporal redundancy* in the working memory and *structural similarity* in the production memory in order to avoid examining the whole of these memories in each cycle. *Temporal redundancy* expresses the fact that not all elements in working memory change in each cycle and that any production that becomes instantiated was close to being instantiated in the previous cycle. *Structural similarity* expresses the fact that many productions have many conditional elements in common in their LHS. Based on these two properties, the Rete algorithm, instead of examining the working memory directly, monitors the changes made to working memory and maintains internal information that is equivalent to the working memory. At the beginning of a cycle the match routine computes whether any changes need to be made to the conflict set. If there are changes to be made, it sends these changes to the interpreter where the conflict set is being maintained.

The interpreter consists of a fixed part that deals with the conflict set and a variable part that is generated by the compilation of the LHS of the productions into Rete networks. These networks perform the actual match. Structural similarity is achieved in them by combining nodes that test for the same LHS conditional elements.

For each working memory element, a token is created for the pair of the working memory element and a tag. The tag is used in order to determine whether the working memory element is added or removed from the working memory. Tokens are processed by the nodes in the network in order to determine whether the overall pattern is matched or not.

Example. The following productions contain two identical condition elements:

$$\begin{aligned} MB15 \quad & ((Want\ (Monkey\ On\ ?O))(?O\ Near\ ?X) \Rightarrow (Want\ (Monkey\ Near\ ?X))) \\ MB16 \quad & ((Want\ (Monkey\ On\ ?O))(?O\ Near\ ?X)(Monkey\ Near\ ?X) \Rightarrow \\ & (Want\ (EmptyHanded\ Monkey))) \end{aligned}$$

When these two productions are compiled together most of the nodes in the network are shared. Figure 2.1 describes the Rete algorithm for the two productions above.

Jess. Jess [51] is a java shell that is based on CLIPS [19] which has been developed by the Technology Branch (STB) of NASA. CLIPS implements a forward chaining rule interpreter that cycles through a match-execute cycle. During the matching phase, all rules are scanned to determine whose condition part is matched by the current state of the environment, i.e., contents of working memory. During the execute phase the operation specified by the action part of the rule that has been found to match is executed. This cycle continues until either no rule matches or the system is explicitly stopped by the action part of a rule. A conflict resolution strategy is applied if more than one rule is found to match. The environment is modelled through a set of "facts" which are kept in a list in memory.

Planning Systems. Planning systems [74] are systems that given a *formula* that represents a *goal* γ they attempt to find a sequence of *actions* that produces a *world state* described by some state description S such that $S \models \gamma$. We say then that the state description satisfies the goal. Although the *state-space*

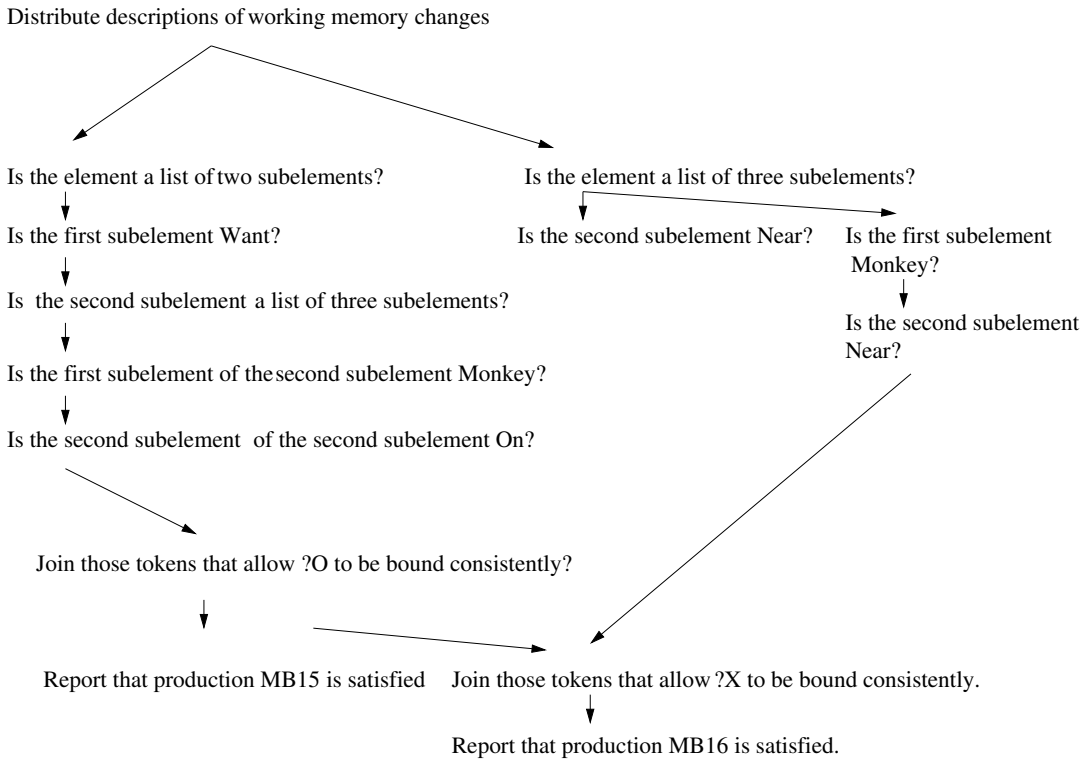


Figure 2.1. The Rete algorithm for productions MB15 and MB16.

approach of planning systems *looks promising* for Sentient Computing, the goals pursued by planning systems are different, and a lot of work needs to be done in this direction to determine any potential contributions in this area.

2.9 Distribution

This section discusses the current principles in designing distributed systems that are used as guidelines in this dissertation.

ITU. The *ITU-T Recommendation for Open Distributed Processing* [47] is a set of standards according to which distributed systems should be designed. ODP proposes an object modelling approach to system specification and defines a framework for building distributed systems that abides by the principles of *transparency* and *conformance*.

The *ODP model* is composed of *objects*. Objects are representations of entities in the real-world. A system is composed of interacting objects. The *state* of an object is determined by a set of *actions* it can take part in. A subset of actions defines the object's interaction with other objects.

The ODP framework, as part of its event-based model provides an *event notification function* [49] that coordinates interaction between the framework entities. The event notification function specifies *event histories* as objects that represent significant actions and therefore object states. *Event producers* interact with the event notification function to create event histories. The event notification function notifies event consumer objects of the availability of event histories. The event notification function supports one or more *event history types* and has an *event notification policy* which determines the behaviour of the function, in particular,

- which objects can create event histories,
- which objects are notified of the creation of a new event history,
- the means by which such notifications occur,
- persistence and stability requirements for event histories.

An *event consumer* interacts with the event notification function to register for notification of new event histories. Depending upon the event notification policy, the interaction can

- establish bindings to currently available event histories.
- enable communication about event histories created subsequent to the interaction.

2.10 The Object Management Group (OMG)

The Object Management Group is a non-profit consortium created in 1989 with the purpose of promoting the theory and practice of object technology in distributed computing systems. In particular, it aims to reduce the complexity, lower the costs, and hasten the introduction of new software applications. Originally formed by 13 companies, OMG membership has grown to over 500 software vendors, developers and users.

OMG realizes its goals through creating standards that allow interoperability and portability of distributed object oriented applications. They produce specifications that are put together using contributions of OMG members who respond to Requests For Information (RFI) and Requests For Proposals (RFP). The strength of this approach comes from the fact that most of the major software companies interested in distributed object oriented development are OMG members.

2.10.1 The Object Management Architecture (OMA)

OMA is a high-level vision of a complete distributed environment. It consists of system-oriented components (Object Request Brokers and Object Services) and application-oriented components (Application Objects and Common Facilities). The Object Request Broker constitutes the foundation of OMA. It allows objects to interact in a heterogeneous, distributed environment, independent of the platforms on which these objects reside and techniques used to implement them. In performing its task, it relies on Object Services, which are responsible for general object management such as creating objects, access control, keeping track of relocated objects, etc. Common Facilities and Application Objects are the components closest to the end user, and in their functions they invoke services of the system components.

2.10.2 The Common Object Request Broker (CORBA)

CORBA specifies a system that provides interoperability between objects in a heterogeneous, distributed environment and in a way transparent to the programmer. Its design is based on the OMG Object Model. The OMG Object Model defines common object semantics for specifying the externally visible characteristics of objects in a standard and implementation-independent way. In this model, clients request services from objects (which will also be called servers) through a well-defined interface. This interface is specified in OMG IDL (Interface Definition Language). A client accesses an object by issuing a request to the object. The request is an event, and it carries information including an operation, the object reference of the service provider, and actual parameters (if any). The object reference is an object name that defines an object reliably.

2.11 Knowledge Integration Systems

This section aims to illustrate the conceptual similarities between the modelling approach presented in this dissertation and the ERP design model. SAP [105] is the most influential ERP system. Though specific to corporate knowledge integration, SAP is based on a design that is platform independent, scalable, highly customisable and integratable. SAP has achieved notable success.

SAP. SAP the company was founded in Germany in 1972 by five ex-IBM engineers. SAP stands for Systeme, Andwendungen, Produkte in der Datenverarbeitung which - translated to English - means Systems, Applications, Products in Data Processing. In 1979, SAP released SAP R/2 (which runs on mainframes) into the German market. SAP R/2 was the first integrated, enterprise wide package and was an immediate success. For years SAP stayed within the German borders until it had penetrated practically every large German company. Looking for more growth, SAP expanded into the remainder of Europe during the 80's. Towards the end of the 80's, client-server architecture became popular and SAP responded with the release of SAP R/3 (in 1992). This turned out to be a "killer application" for SAP, especially in the North American region into which SAP expanded in 1988.

SAP R/3 is a highly integrated, highly scalable system that models corporate processes that extend through various departments. It is built on robust databases, and it supports different database technologies and operating system platforms. This makes it open. It offers support for knowledge persistence, archiving and report production. SAP is delivered to a customer with selected standard processes turned on and many other optional processes and features turned off. This makes it highly customisable and portable. At the heart of SAP R/3 are about 10,000 tables that control the way the processes are executed. Configuration is the process of adjusting the settings of these tables to get SAP to run the way you want it to. Functionality included is enterprise-wide including: Financial Accounting (e.g., general ledger, accounts receivable etc), Management Accounting (e.g., cost centres, profitability analysis etc.), Sales, Distribution, Manufacturing, Production Planning, Purchasing, Human Resources, Payroll, etc. All of these modules are tightly integrated.

SAP are maintaining and increasing their dominance over their competitors through a secure, personalisable and customisable entry point in the knowledge system (an Internet portal), which allows different privileges for different users. Furthermore, it embraces legacy systems such as mainframes by offering generic middleware wrappers that offer seamless integration. The Business Connector is such a component.

2.12 Security

Finally, related work has been carried out in the area of security for context-aware systems. Two efforts that have had a significant impact in the area of context-aware computing are discussed here.

Spatial Policies for Mobile Agents in a Sentient Computing Environment. The work presented in [99] discusses a simple location-based mechanism for the creation of security policies to control mobile agents. It simplifies the task of producing applications for a pervasive computing environment through the constrained use of mobile agents. The novelty of this work lies in the fact that it demonstrates the applicability of recent theoretical work using ambients [14] to model mobility.

This work can be integrated with the model presented in this dissertation as an additional service to control the behaviour of mobile agents according to the defined spatial policies.

Location Privacy in Sentient Computing. The work presented in [11] discusses the issue of maintaining user privacy and particularly user anonymity in Sentient Computing. In such an environment, a user's anonymity can be breached and his identity be inferred by a number of factors, such as the fact

that he spends most of his time at his desk. This is addressed by applying the principle of a mixed zone which is an area where none of the users has registered an application callback and therefore is anonymous. In a mixed zone, a previously identifiable user is dispersed among the rest of the users. Assuming that users change to a new, unused pseudonym whenever they enter a mixed zone, applications that see a user emerge from the mixed zone cannot distinguish that user from any other who was in the mixed zone and cannot link people going into the mixed zone with those coming out of it.

This work complements directly the work described in Chapter 3 of this dissertation where a number of possible inferencing mechanisms based on location are discussed and which demonstrate a clear threat to user anonymity. This gives good grounds for the findings of [11] to be applied.

Chapter 3

Inferring Abstract State from Concrete State using Hidden Markov Models (HMMs)

This chapter discusses a methodology for modelling high-level abstract knowledge that is inferred from low-level concrete knowledge such as location data. The inference process is based on *Hidden Markov Models (HMMs)* [32, 88]. This chapter uses this methodology in order to build a recognition system for *movement models*, here referred to as *movement phonemes*. The recognition system is used as a middleware component in SCAFOS and movement phonemes are used by the user in order to create application specifications.

3.1 Introduction

The specific aim of SCAFOS is to create *models* of abstract knowledge. Such models are to be made available to the user who can use them in order to create application specifications using SCAFOS. Because the user is mobile and the world consists of several heterogeneous distributed domains which may change dynamically, the abstract models should be valid independent of the topology of the physical environment and independent of the user. The benefits of having models of abstract knowledge with the above properties can be summarised below:

- Models of abstract state that are independent of the specifics of each sensor-driven component (e.g., with respect to the number of users and the topology of the environment) can be applied to heterogeneous sensor-driven systems and still remain valid.
- The validity and accuracy of such models in different sensor-driven environments can be used to evaluate the properties of the sensor technology that produces the model data, such as its accuracy. For example, two different location technologies can be evaluated based on how well they recognise human movements, such as the ones described in this chapter.
- Movement models are potentially capable of letting us learn a great deal about the real-world process that produced the location events without having the source available, i.e., by simulation.

3.2 Achievements

Section 3.1 discusses the motivation of this thesis in creating models for statistical inferencing. In this section, the specific achievements that stem from applying HMMs to the Sentient Computing environment are presented. This chapter achieves the following:

- The design and implementation of a system that automatically infers high-level knowledge from time-series location data (such as those produced by the Active BAT) when it has been appropriately trained. Such a system behaves similarly to a speech recogniser that recognises words from processing speech signals. More specifically:
  - HMMs are shown to be appropriate for characterising location-based inferences.
  - A set of semantics that is appropriate to the Sentient Computing Environment and detectable by HMMs in real-time is determined.
  - Event models that represent the identified semantics are discussed. These models are referred to as *phonemes*.
  - The design of an appropriate sampling process for the training and the observation data streams is discussed.
  - The design and implementation of a system that performs recognition for the above phonemes is discussed.
- The inverse problem to phoneme recognition is investigated, i.e., the differences in the environment that cause differentiations in the patterns that correspond to the same phoneme are identified. More specifically:
  - The design and implementation of a user recognition system is discussed that is based on samples of users sitting down.
- Both recognition systems are evaluated against existing related work.

3.3 Justification

The choice of the models to be recognised was based on the following criteria:

- The models reflect invariant properties that are common among different users and are recognisable independently from the user.
- The chosen invariant properties are reliably and accurately recognisable independently of the user's speed of motion. For example, a walking movement should be recognisable whether a user is walking slowly, with medium speed or quickly. This is not straightforward. In the case of SPIRIT [41], the speed with which a user is moving determines the sampling rate of SPIRIT. This has a direct effect on the sampling method. If samples of equal duration are taken, then samples that represent the same phoneme but which have been produced by users moving at different speeds will have a different number of data points and therefore will be different (see Section 3.6). This issue has been addressed successfully, and it is discussed later on in this chapter.
- The models are *transparent* to the environment, i.e., they are re-applicable to any sensor-driven environment.
- The models are used in order to produce a characterisation of each location event. The characterisation is meaningful in the general case forming a complete set of recognisable phonemes. In some cases, further or different inferencing could also be appropriate. For example, the movement of a user opening a door inwards and entering a room is currently recognisable when compared to walking in a straight line; however, it is not easily distinguishable from turning right or turning left movements. Therefore, a model for opening a door inwards is not made part of a general-purpose recognition system, as it is not recognisable against the complete set of phonemes (see Section 3.10 for discussion).

3.3.1 Phonemes

The following models were constructed as characterising certain 3-D space movements:

- Sit Down
- Stand Up
- Sitting
- Walking
- Still

The pattern in each phoneme reflects the gravity and the displacement of the human body in 3-D space. These are mutually exclusive and comprise a complete set of movements, in the sense that if a user performs a movement that is identified as belonging to one of the above models, this movement describes fully the user's state in terms of moving. Although previous attempts have been made in the area of movement recognition, these have been case-specific and limited. Section 3.8 demonstrates the efficiency of the modelling approach over existing methodologies.

3.4 The Movement Recognition Problem

A *Hidden Markov Model* (HMM) is a stochastic model where an underlying process that is not observable can be observed through another set of stochastic processes that produce the sequence of observations.

An HMM can be seen as a finite state machine that consists of N states denoted as $X = x_1, x_2, \cdots, x_N$ and the state at time t as q_t. An HMM is characterised by the following:

- S, the number of distinct observation symbols per state, i.e., the discrete alphabet size. The observation symbols correspond to the physical output of the system being modelled. We denote the individual symbols as $V = v_1, v_2, \cdots, v_S$.
- The state transition probability distribution $A = a_{ij}$ where

$$a_{ij} = P[q_{t+1} = x_j | q_t = x_i], \quad 1 \leq i, j \leq N$$

- The observation symbol probability distribution in state j, $B = b_j(k)$, for a fixed time t, where

$$b_j(k) = P[v_k \; at \; t | q_t = x_j], \quad \begin{array}{l} 1 \leq j \leq N \\ 1 \leq k \leq S \end{array}$$

- The initial state distribution $\pi = \{\pi_i\}$, i.e./ the probability that each state x_i is the first state

$$\pi_i = P[q_1 = x_i], \quad 1 \leq i \leq N$$

Each time that a state j is entered at time t, an observation vector O_t is generated from the probability density $b_j(O_t)$. After the HMM has moved from the initial state x_0 to a final state x_{T+1} for this sequence, a sequence of observations has been generated: $O = O_1 O_2 \cdots O_T$, where each observation O_t is one of the symbols from V and T is the number of observations in the sequence (it is assumed that states x_0 and x_{T+1} do not produce any observations).

Figure 3.1 shows an example of this process where a six state model moves through the state sequence $X = 1, 2, 2, 3, 4, 4, 5, 6$ in order to generate the sequence O_1 to O_6 (States 1 and 6 are the initial and final states and they do not generate any observations).

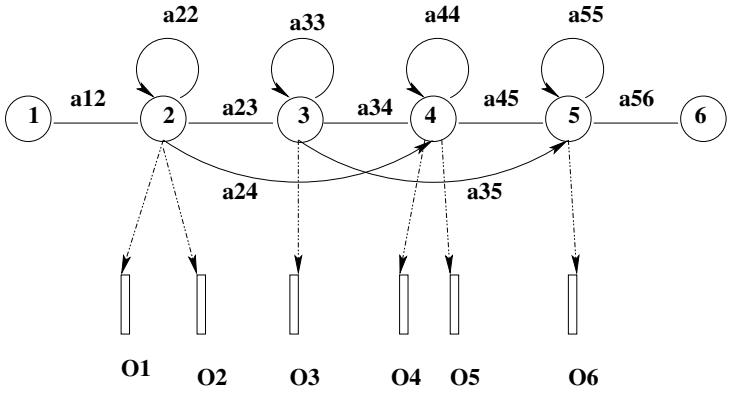


Figure 3.1. The Markov generation model

The general recognition problem can be seen as classifying an observation sequence O_T to the HMM that represents the hidden underlying process that generated the observation sequence. This problem entails three more specific problems: the first is that when trying to create HMM models for each movement phoneme, the values of the state transition probabilities a_{ij} and output probabilities b_j of each model are not known and need to be estimated by training data. The better the estimation, the more accurate the model. The second problem arises when trying to uncover the hidden part of the model. As the process to be modelled (movement phoneme) is unknown, the state sequence that generated an observation is not known either. The third problem is a problem of evaluation: how is the most appropriate model that generated the observation sequence defined, out of a set of possible models?

Assuming a vocabulary that consists of words w_i that represent the phonemes of interest, i.e., $w_i \in \{SitDown, StandUp, Walking, Still, Sitting\}$, let each movement be represented by a sequence of position vectors of three dimensions (x, y, z) or observations O, defined as

$$O = \langle O_1, O_2, \cdots, O_T \rangle \tag{3.1}$$

where O_t is the position[1] observed at time t. The phoneme recognition problem can then be regarded as that of computing the model w_i with the maximum probability of having generated the observation sequence O.

$$\arg\max_i P(w_i|O) \tag{3.2}$$

where w_i is the i^{th} phoneme in the vocabulary.

This probability is not computable directly, but using Bayes' rule gives

$$P(w_i|O) = \frac{P(O|w_i)P(w_i)}{P(O)}. \tag{3.3}$$

Equation (3.2) is solved using (3.3) if $P(O|w_i)$ can be estimated. The general problem of the direct estimation of the joint conditional probability $P(O_1, O_2, ...O_T|w_i)$ from examples of location samples, given the dimensionality of the observation sequence O, is not practicable. However, if a parametric model of word production such as a Markov model is used, then estimation from data is possible since the problem of estimating the class conditional observation densities $P(O|w_i)$ is replaced by the mathe-

[1] O_t is a vector of three variables x, y, z that represent the coordinates.

matically much simpler problem of estimating the Markov model parameters, which entails significantly smaller computational effort. Given an HMM model, the joint probability that O is generated by the model M moving through the state sequence X of Figure 3.1 is calculated simply as the product of the transition probabilities and the output probabilities:

$$P(O, X|M) = a_{12}b_2(O_1)a_{22}b_2(O_2)a_{23}b_3(O_3)\cdots \tag{3.4}$$

Given that X is unknown, the required likelihood is computed by summing over all possible state sequences $X = x(1), x(2), x(3), \cdots, x(T)$, that is

$$P(O|M) = \sum_x a_{x(0)x(1)} \prod_{t=1}^{T} b_{x(t)}(O_t)a_{x(t)x(t+1)} \tag{3.5}$$

where $x(0)$ is the model's entry state and $x(T+1)$ is the model exit state.

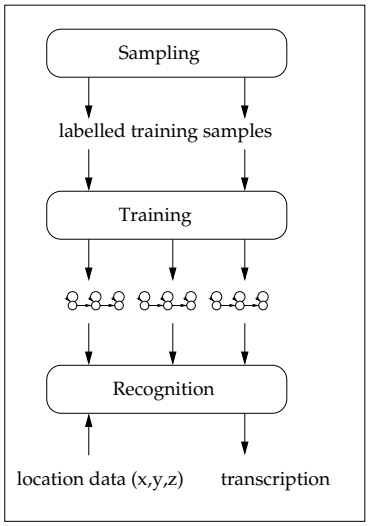


Figure 3.2. Phoneme recognition

As an alternative to Equation 3.4, the likelihood can be approximated by only considering the most likely state sequence that is

$$\hat{P}(O|M) = max_x\{a_{x(0)x(1)} \prod_{t=1}^{T} b_{x(t)}(O_t)a_{x(t)x(t+1)}\} \tag{3.6}$$

This assumes that the parameters a_{ij} and b_j are known. Although this is not generally the case, HMMs allow for the above parameters to be estimated using training data. This process is called *training* (Figure 3.2).

Training. Given a set of training examples corresponding to a particular model, the parameters of that model can be determined automatically by a statistically robust and efficient re-estimation procedure (*Baum-Welch re-estimation.* This procedure has the following steps:

A set of prototype models are created, in which the output distribution for each state j is Gaussian

with mean vector μ_j and covariance matrix Σ_j; that is, $b_j(O_t)$ satisfies:

$$b_j(O_t) = N(O_t; \mu_j, \Sigma_j)$$

If $L_j(t)$ denotes the probability of being in state j at time t, then the maximum likelihood estimates of μ_j and Σ_j can be calculated as shown below

$$\hat{\mu}_j = \frac{\sum_{t=1}^{T} L_j(t) O_t}{\sum_{t=1}^{T} L_j(t)}, \tag{3.7}$$

$$\hat{\Sigma}_j = \frac{\sum_{t=1}^{T} L_j(t)(O_t - \mu_j)(O_t - \mu_j)'}{\sum_{t=1}^{T} L_j(t)}, \tag{3.8}$$

where prime denotes vector transpose. To apply the above equations, the probability $L_j(t)$ must be calculated. This is done efficiently using the *Forward-Backward* algorithm. Executing the above produces a set of models, which are optimised according to the training data.

Recognition. Recognition of an unknown data sample of size s is based on building an HMM network and finding a path of size s that has the maximum likelihood (Viterbi algorithm). That path corresponds to the HMM model that corresponds to the correct phoneme. The model with the highest maximum likelihood is selected for each observation sequence under consideration (Token Passing Algorithm).

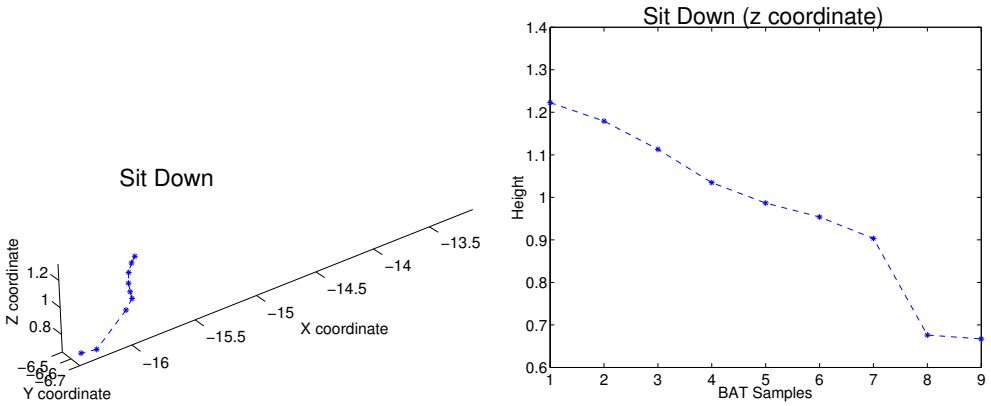
3.5 Building Phoneme Models

This section portrays a representation of the phoneme models discussed in Section 3.3.1 in terms of their observations being 3-D position vectors (x, y, z). The coordinate system is arbitrarily set in order to depict user positions in the LCE [36] and is currently being used in SPIRIT [41] and its applications. The horizontal axis in most of the graphs that follow is the x axis and the vertical axis is the y axis in this coordinate system. The z coordinate signifies the distance from the user's Active BAT to the floor. On the graphs that portray only one coordinate on the vertical axis, the horizontal axis always portrays the number of sampling points since the beginning of the sampling process.

The models presented next are based on a number of experiments that took place in the LCE. The samples for the Sit Down phoneme were produced by a user sitting down and getting up from a couch. The height when standing up (z coordinate) is approximately $1.2\ m$. Several other experiments took place and the results are discussed in Section 3.9.

3.5.1 Sit Down

The sit down movement is characterised by the fact that the user bends his knees, lowering his body onto the seat. During this movement, the user leans slightly forward and then backwards onto the seat.



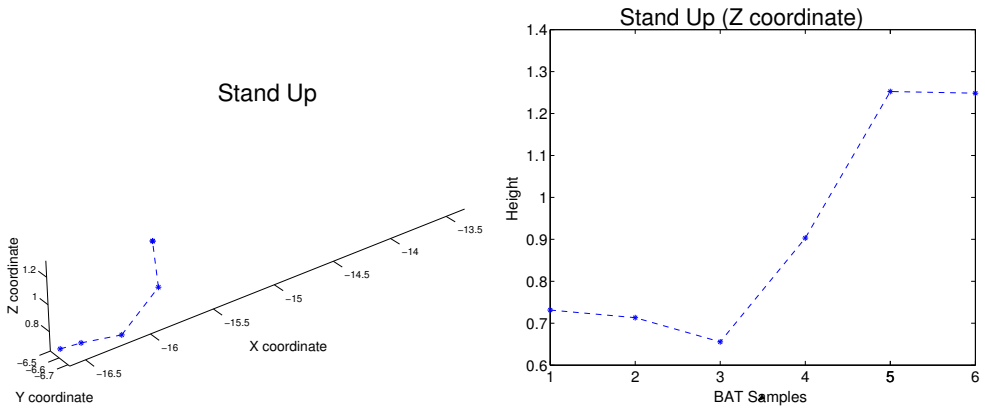
(a) (b)

Figure 3.3. A Sit Down sample 3D (a), z coordinate only (b).

A sample of a sit down movement (on the LCE's couch) is portrayed in Figure 3.3. The characteristic sample (b) is produced by the z coordinate as captured by the Active BAT which is portrayed on the abscissa of Figure 3.3(b).

3.5.2 Stand Up

A stand up movement is characterised by the fact that the user drags his body slightly forwards, until the feet are placed strongly on the floor, bends the knees lowering his body forward and straightens the knees gradually, thus lifting the body upwards.



(a) (b)

Figure 3.4. A Stand Up sample 3D (a), z coordinate only (b).

Figure 3.4 shows a Stand Up sample, as monitored by the Active BAT.

3.5.3 Sitting

The sitting state is characterised by the fact that the user remains still with the body closer to the ground than when standing up.

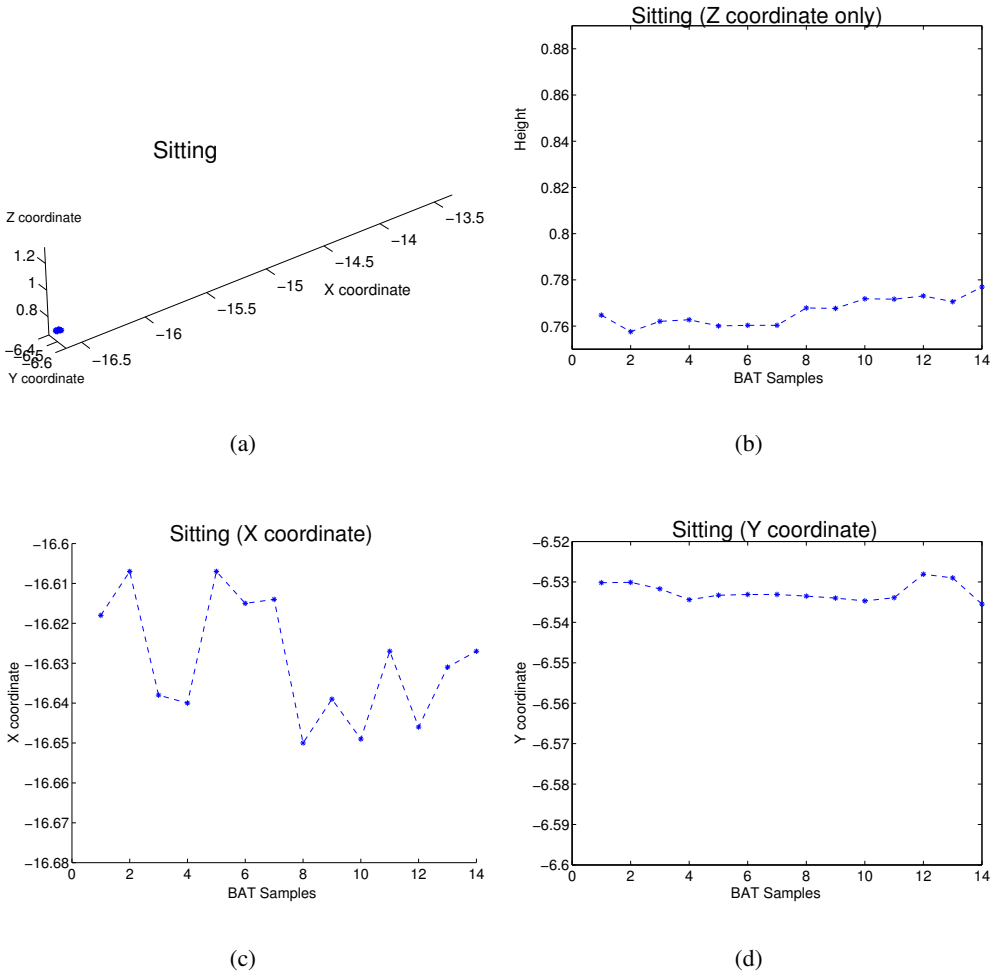
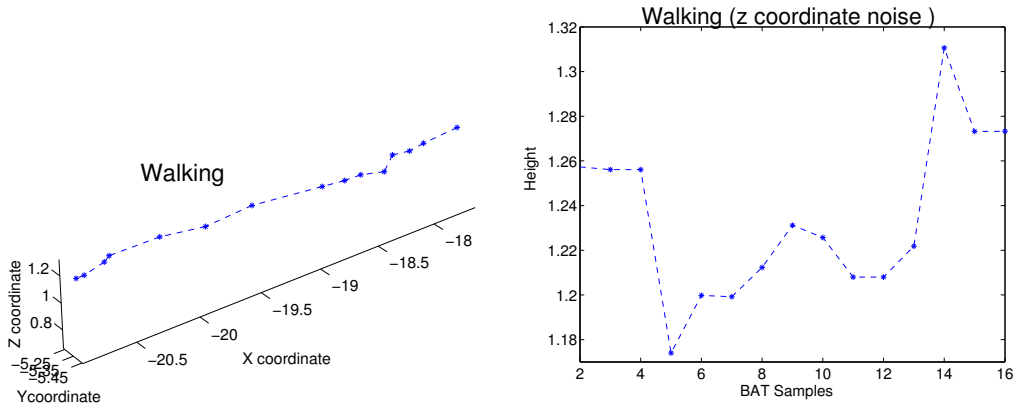


Figure 3.5. A Sitting sample 3D (a) z coordinate only (b) x coordinate only (c) y coordinate only (d).

A sitting sample is portrayed in Figure 3.5. The characteristic sample (b) is produced by monitoring the z coordinate. Although during the experiment the user was still, there is a small displacement in the sample points. This is due to the statistical error of the sampling process, and it is obvious also in plots (b), (c) and (d), which portray each coordinate separately vs. the sample number.

3.5.4 Walking

The walking movement is characterised by a fluctuation in the user's height. As he lifts each leg and lowers it again his whole body is displaced slightly up and down.



(a) (b)

Figure 3.6. A Walking sample 3D (a) z coordinate only (b).

A sample of a user walking is portrayed in Figure 3.6. The characteristic sample (b) is produced by monitoring the z coordinate. Although there is a similar displacement in the x and y coordinates as well, this is not always obvious as this displacement may be due to the user turning, or walking in a curve. For this reason, the plots of the x and y coordinate are not considered characteristic for this phoneme and are not portrayed here.

3.5.5 Still

The still state is characterised by the fact that the position of the user remains the same in all three coordinates.

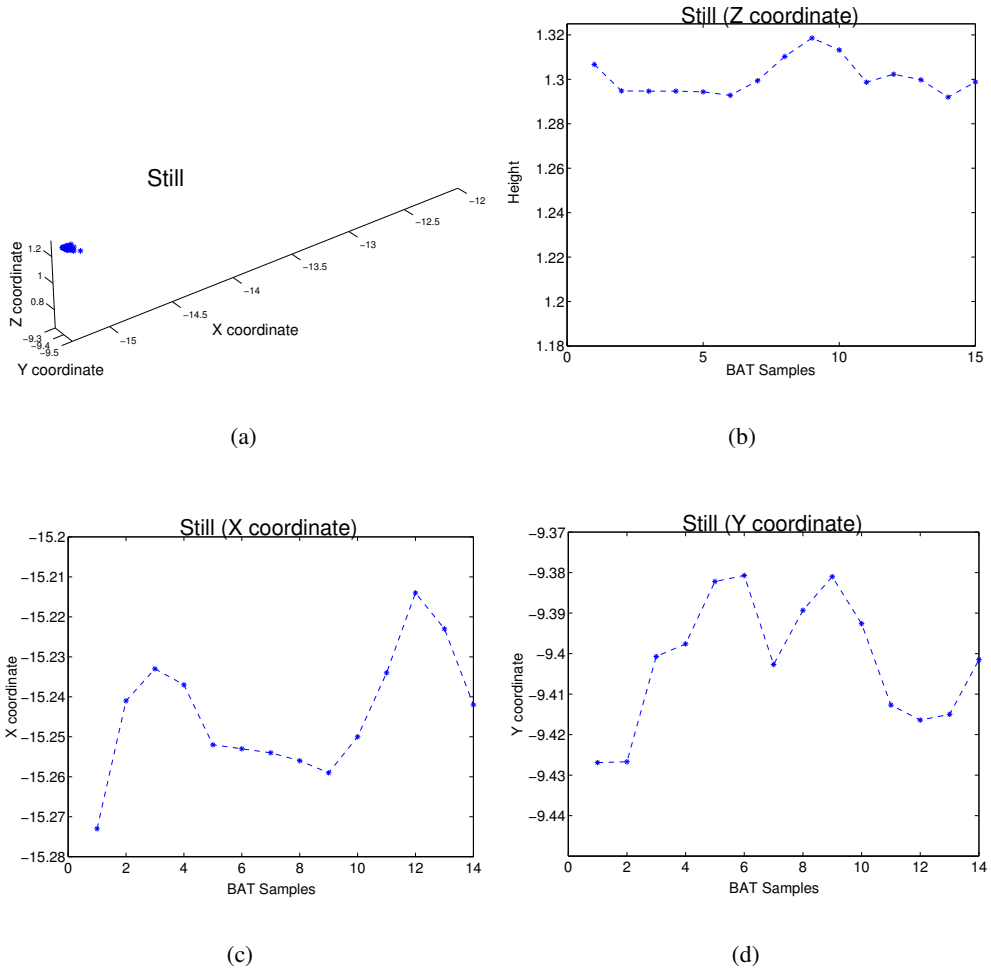


Figure 3.7. A Still sample 3D (a) z coordinate only (b) x coordinate only (c) y coordinate only (d).

A sample of a user who is still is portrayed in Figure 3.7. Also in this case, there is an obvious displacement in all coordinates that is caused by the error in the measurements. The shape of the z coordinate in the sample in Figure 3.7(b) bears similarities to the sample of Figure 3.6(b). However, the latter has a more distinguishable shape and the displacements are on average larger than the ones of Figure 3.7(b). This similarity in shape creates ambiguity between the two phonemes when trying to recognise them with traditional methodologies. The HMM-based solution does not suffer from the same deficiency. This is discussed in detail in Section 3.8.

3.5.6 Open Door Outwards

The Open Door Outwards phoneme is characterised by the fact that the user pulls the door outwards walks around it in a curve and enters the opening inwards.

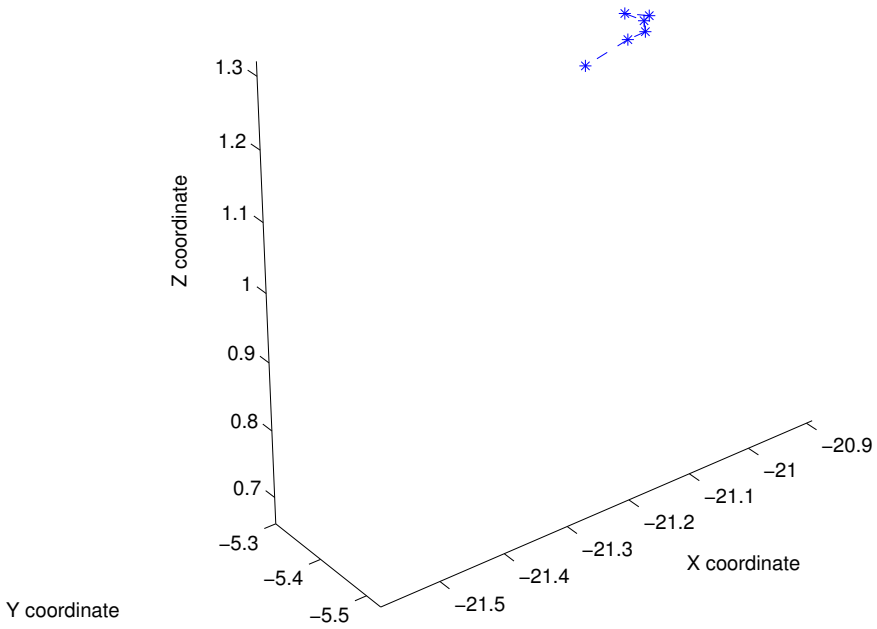


Figure 3.8. A sample of movement patterns entering doors that open outwards.

Although this phoneme was not selected to form part of the final recogniser, it is discussed here in order to illustrate the potential of using HMMs to discover aspects of the physical environment.

3.5.7 Walking-Still-Sit Down-Stand Up

A user walks to the couch, sits down, remains seated for a while, stands up and remains still.

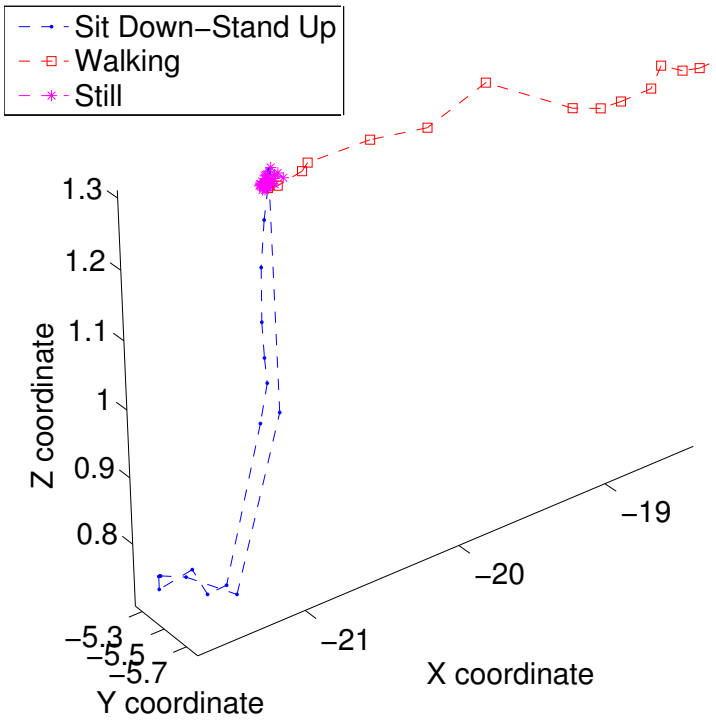


Figure 3.9. All phonemes.

Figure 3.9 portrays a sequence of movements. This aims to give an illustrations of a movement that consists of multiple samples.

3.6 Identifying an Appropriate Sampling Process

The samples used for the phonemes in Section 3.5 are position vectors (x, y, z) of variable length, and they are portrayed as produced by the experiment. This section deals with identifying an appropriate sampling process that can be undertaken automatically by a sampling service in real-time.

Two factors need to be taken into consideration. The first one is the *speed* of the executed movement. The second is the *sampling rate* of the location technology. As far as the first factor is concerned, it is desirable that movements should be recognised independently of the speed with which they are executed, i.e., quick movements should be as reliably recognisable as slow ones. The second factor affects the sampling decision only when the rate at which positions are sampled by the sensor infrastructure depends on the user speed. SPIRIT is such a system that employs the principle of a variable sampling rate for sampling efficiency. Wherever it is determined from the location samples that an object is increasing its speed, SPIRIT increases the Active BAT's sampling rate too. On the contrary, if an object slows down, SPIRIT decreases the Active BAT's sampling rate.

In order to decide the best sampling process, samples of both fixed duration (6 seconds) and fixed length (6 observations and 4 observations) were considered. Samples of fixed duration contained a variable number of data points, and in order to be used as training sequences and be recognised by the HMMs, an *interpolation* process was used where the missing data points were replaced with a copy of the value of the last observation in the sample. Although the recognition was similar in accuracy to that achieved using fixed-length samples (see Section 3.6.4), it was decided to use fixed-length samples for simplicity reasons, in order to avoid the interpolation overhead. Section 3.6.3 discusses how samples are distinguished and recognised under this sampling scheme.

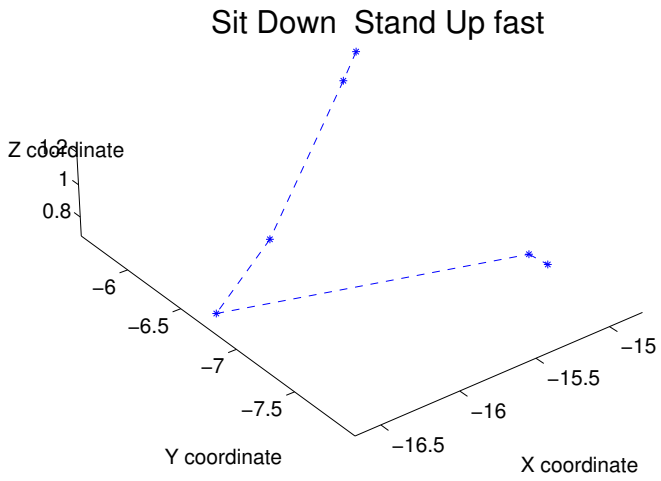
The following section proposes a different calibration process, appropriate for movement recognition, so that SPIRIT's variable sampling rate does not affect the accuracy of the recogniser.

3.6.1 Movement Calibration

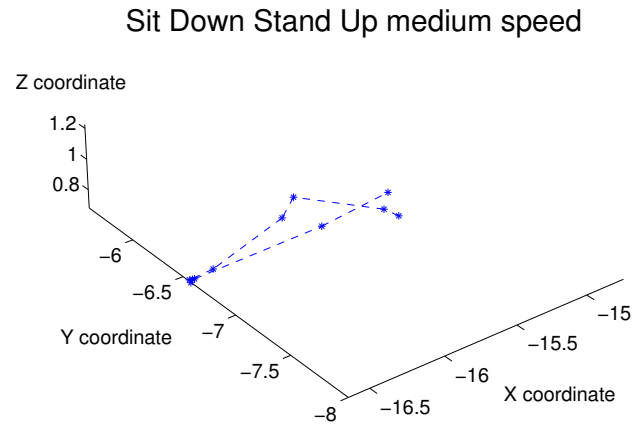
Given SPIRIT's variable sampling rate, it is obvious that the worst-case sample to be recognised is one where the user's speed is high and the sampling rate of the location system is low. It was decided that the calibration should be performed using the Sit Down and Stand Up movements, as these have a more distinctive shape to the human eye than the Walking and Still patterns, and the beginning and end of the movement can be simulated easily. In order to perform this calibration, the following experiment was carried out.

First a user performed twenty sit down and stand up movements on the couch in the lab, at different speeds. The fastest sit down movement produced a sample consisting of 4 events (data points) and lasted 1.032 sec in total. The fastest stand up movement consisted also of 4 data points. All other samples consisted of more data points. An appropriate technique in HMMs is that the sampling window is set to 4. Larger samples are processed as a series of samples of size 4 (e.g., one sample of size 15 is seens as 12 samples of size 4.) Figure 3.10 portrays some representative samples of duration 20 sec of sitting on the couch at different speeds. Note that because the fast samples were taken at the beginning of the experiment and after the user has been standing still for a while, SPIRIT is sampling with low frequency, which explains why the first sample has only 6 data points. Already at the second fast movement, SPIRIT is sampling at high rate (13 data points). The third sample at medium speed has the most number of data points for the same duration (19 data points), because SPIRIT has adjusted the sampling rate and the movement is slower, so more points are captured per movement.

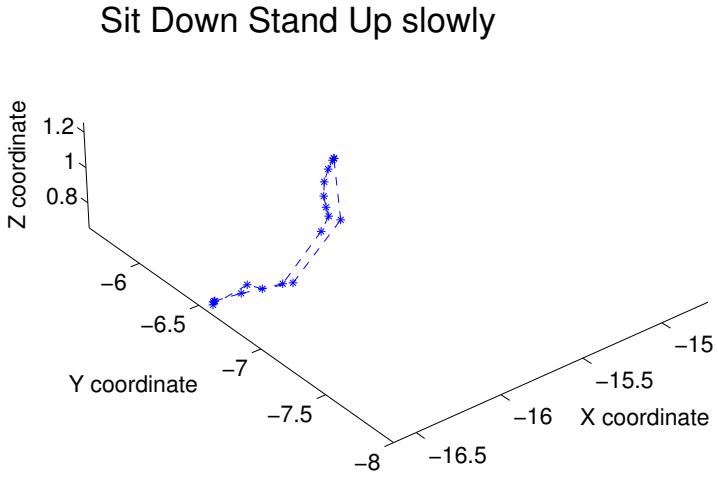
Samples of similar size are often encountered in problems of classifying discrete data and are classified very accurately. For example, the samples that correspond to trips between significant locations [4] often consist of observations of length three and dimensionality one, i.e., 3 consecutive GPS readings. An HMM-based system that calculates user locations based on signal strength measurements is based on



(a)



(b)



(c)

Figure 3.10. Three selected samples of variable length of (a) sitting on couch fast (6 points) (b) at medium speed (13 data points) (c) slowly (19 data points).

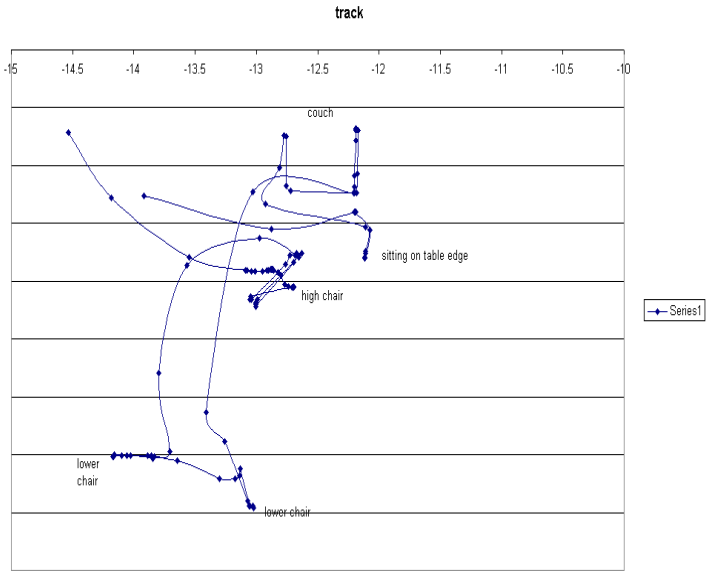
Phonemes	Training Samples
Stand Up	6
Sit Down	8
Sitting	22
Walking	43
Still	44
Total	123

Table 3.1. Size of the training set.

observation sequences of length one and dimensionality four, i.e., each location is modelled by a single signal strength reading from 4 sensors that correspond to that location.

3.6.2 Supervised Learning

The training phase requires a set of representative training samples. For this reason a number of samples were collected by means of *supervised learning*. Figure 3.11 portrays a supervised learning process whereby samples were gathered from a user sitting down on different chairs in the LCE meeting room. Figure 3.12 portrays a supervised learning process for Still samples. Table 3.1 portrays the overall number of training samples for all phonemes.



(a)

Figure 3.11. Supervised learning for Sit Down traning samples (x, y coordinates).

3.6.3 Real-Time Windowed Sampling

Having determined the sample size to be four, a windowing system with a window size of four that produces samples from the stream of positions was required. In this system, the window acts as a

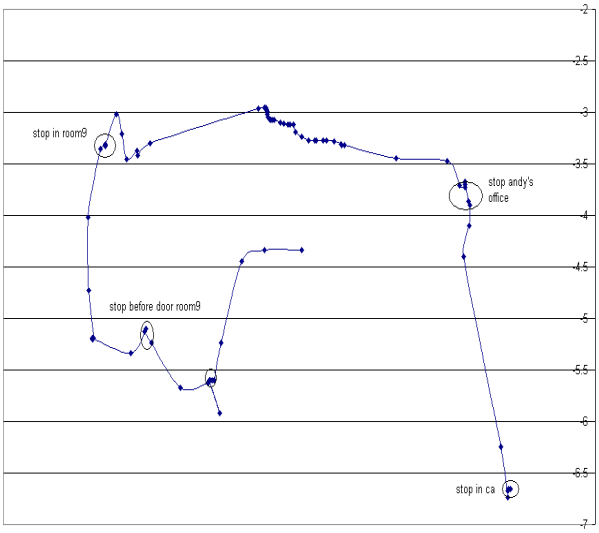


Figure 3.12. Supervised learning for Still training samples (x, y coordinates).

buffer of three data points. Each new event produced by the location system is tested against the three previous ones, which are buffered until the movement is detected, and, by means of the *word network* of Figure 3.14, it is decided to which phoneme the data point belongs. Figure 3.13 illustrates how the phonemes Walking, Sit Down, Sitting and Stand Up are delimited with the help of the window. The phoneme *Still* is delimited from Walking, Stand Up and Sit Down in a similar manner.

Figure 3.15 portrays three training samples of sitting down. The training data was carefully chosen in order to train the recogniser regarding how phonemes should be delimited. For example, it can be seen from Figure 3.11 which samples are characterised as Sit Down with respect to the pattern formed by the data points of the z coordinate. Note that because the Sit Down phoneme may consist of more than 4 data points, more than one consecutive sample can be characterised as Sit Down. For example, the sample portrayed in 3.15(d) can be sampled immediately after 3.15(b) or 3.15(f). This means that as long as one of the three consecutive samples (b), (d) or (f) is recognised correctly, the information about the user sitting down will not be lost. Similarly, Figure 3.16 portrays two selected training samples for the Stand Up phoneme.

3.6.4 Implementation

The recognition problem presented in Section 3.4 was implemented by building a recogniser for movement phonemes, based on HMMs. The HTK toolkit was used for this purpose. The following solutions were implemented:

- Distinguish between Walking, Still, Sitting, Sit Down and Stand Up for the same user.
- Distinguish between Walking, Still, Sitting, Sit Down and Stand Up for different users while the models have been trained only for one.

3.6.5 Recognition Scores

The results file from the recogniser has then been compared against a reference file where the same samples were correctly labelled and a recognition score is assigned to the recognition as a *percentage of correct labels over the overall number of labels*.

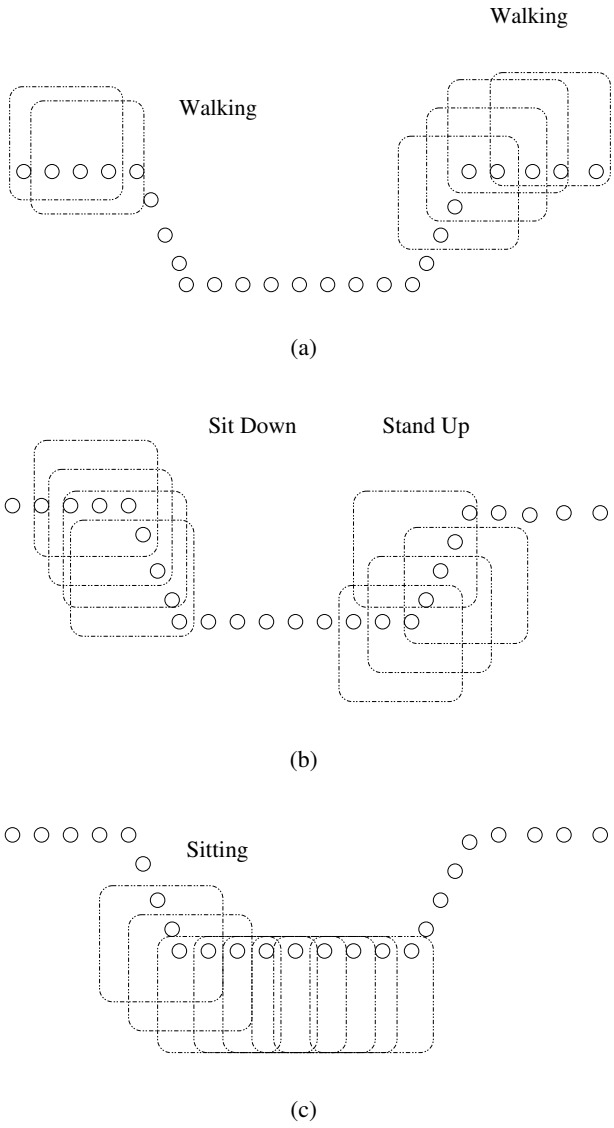


Figure 3.13. Phoneme delimiting.

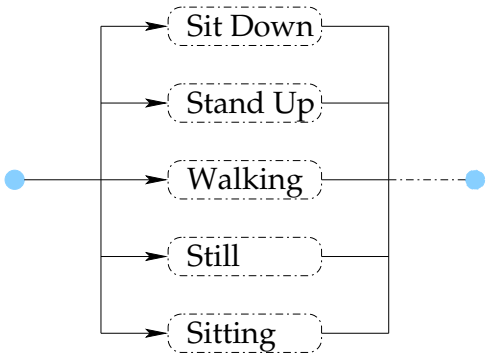


Figure 3.14. The word network for movement phonemes.

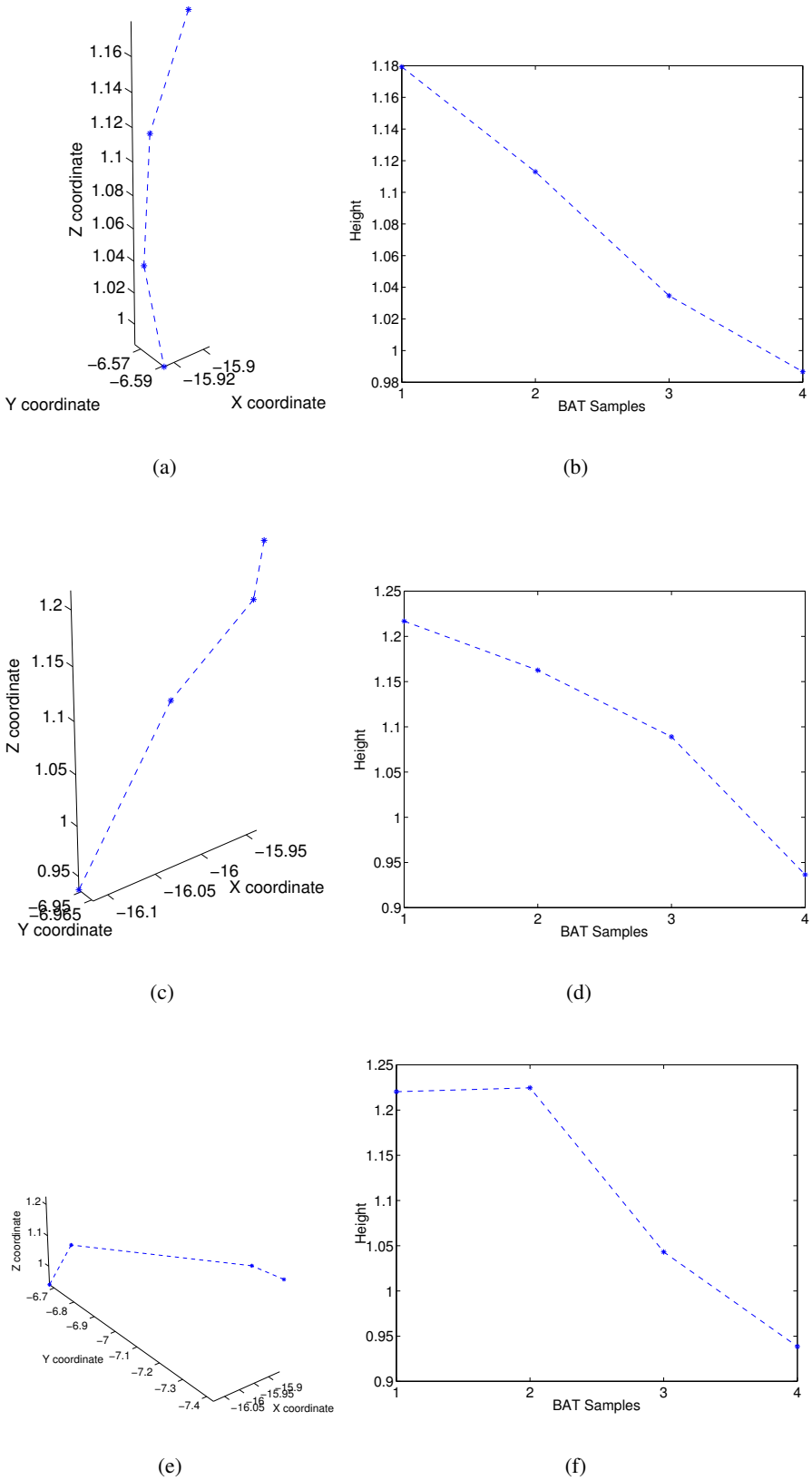


Figure 3.15. Three selected training samples of the Sit Down phoneme with sample size 4.

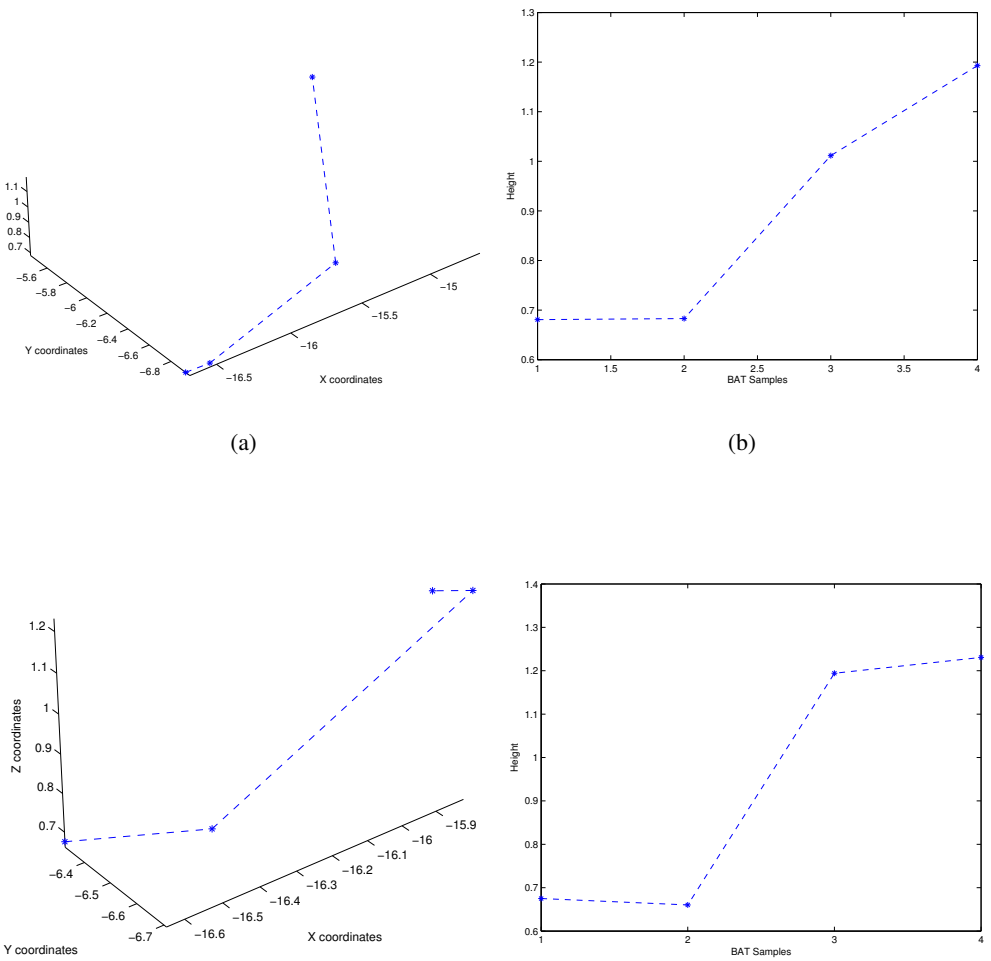


Figure 3.16. Two selected Stand Up training samples.

Phonemes	Training Samples	Test Samples	Correct	Recognition score
Walking-Still-Sitting-Sit Down-Stand Up (same user)	123	46	43	93.18%
Walking-Still-Sitting-Sit Down-Stand Up (two users)	123	45	41	91.11%
Open Door Outwards - Walking (same user)	21	9	7	77.78%
User Recognition (two users)	31	8	8	100%

Table 3.2. Recognition scores.

Table 3.2 contains the recognition score for the movement phonemes tested for the same user that produced the training samples as well as an additional user. The recognition score is 93% of the phonemes identified correctly. If recognition is performed for a different set of phonemes, namely, only patterns of doors opening outwards as opposed to walking straight, the recognition score is 77.78%.

3.7 The Discrimination Problem: User Recognition

This section investigates the inverse problem to movement recognition, that is user recognition. The drive behind this goal has been the observation that a user sitting down on seats with different heights causes the production of different tracks by the monitoring system; the question is whether the differences were significant enough in order to identify the chair from the differences in track (Figure 3.18). This can be seen as a *discrimination problem.*

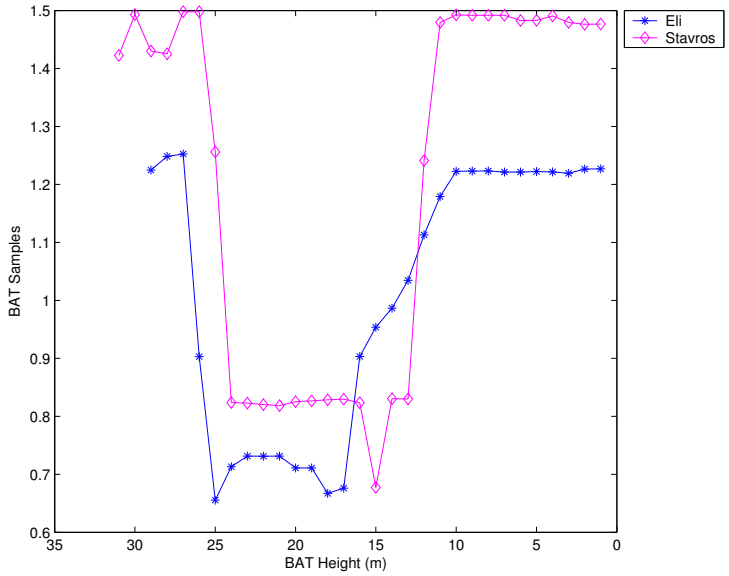
A *user recognition problem*, which consists of distinguishing a user by the patterns produced while sitting down and standing up again, was implemented using the recogniser of Section 3.6.4. The results were very encouraging. Figure 3.17 portrays the tracks of two users of significant differences in height that are used as training samples to a user recogniser. The recognition score for this experiment was 100% (see Table 3.2).

3.8 Technical Background

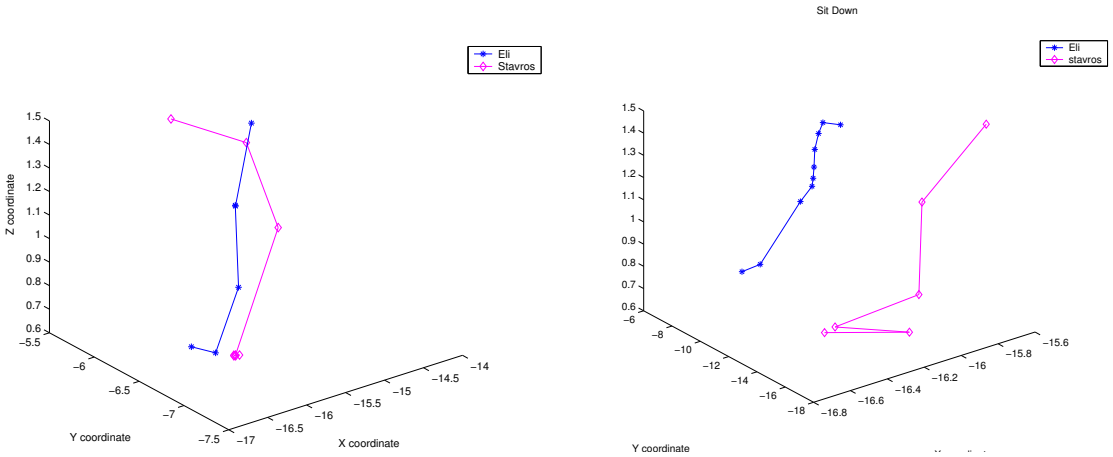
This section discusses HMMs with respect to existing methodologies in the area of movement recognition. Prior work in the SPIRIT [41] system has looked into initial stages of movement recognition in trying to identify when the user is sitting rather than standing by *height calibration*, and when the user is walking by means of *step detection*. The emphasis in this thesis is a toolkit in which users/applications can program models for movements that are recognised independently of domain and user, rather than use hard-coded solutions that are only applicable under specific circumstances.

The most important deficiency of other techniques is that they do not work independently of the user and the implementation domain. On the contrary, they are closely tailored to specific cases and often cannot recognise a movement that is executed under different circumstances.

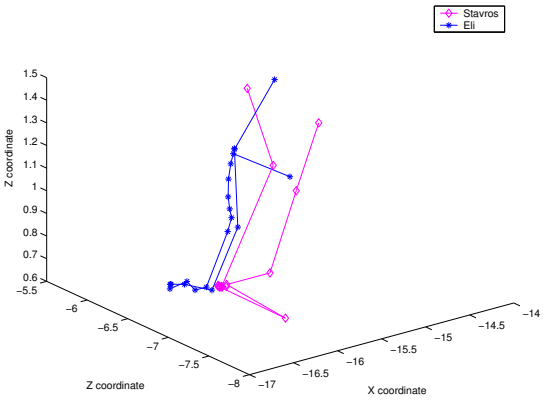
Height Calibration. *Height calibration* is based on a threshold that is calculated according to each user's height. When a position is calculated by SPIRIT where the z coordinate falls below this threshold, the user is assumed to be sitting, and above this threshold to be standing. This method has the disadvantage that any action that causes the BAT to drop below a certain threshold is translated as a Sit Down event. In fact the user might be squatting momentarily or sitting on a tall stool or may have adjusted the cord of his BAT to a different height. This is not distinguishable in SPIRIT. Figure 3.18 portrays the



(a)



(b) (c)



(d)

Figure 3.17. User recognition from their different Sit Down patterns.

changes in the z coordinate of a user sitting on different chairs including a couch. An obvious problem that stems from this figure is where should the threshold be that identifies a user who is sitting down? For example, if the threshold is set to 1 m, then a user sitting on the meeting room table (see right-hand-side sample of Figure 3.18) is not detectable by the Active BAT. On the other hand, the Active BAT is not fixed on the user's body and can be carried at different heights hanging from the neck by a cord of adjustable length or attached to the waist. This affects the choice of threshold, as does the variability of user height. Finally, the height threshold depends also on the position of the sensors on the ceiling, and this, in general, varies from room to room and from building to building. All the above reasons make it impossible to chose a single threshold that can be used to recognise the Sit Down and Stand Up movement under all circumstances. Lastly, using height calibration, it is impossible to discriminate between different sitting tracks and thus recognise users or the object upon which they are sitting. Indeed, using HMMs, the height at which a badge is worn by a particular user can be calibrated by recognising the end of a Stand Up phoneme.

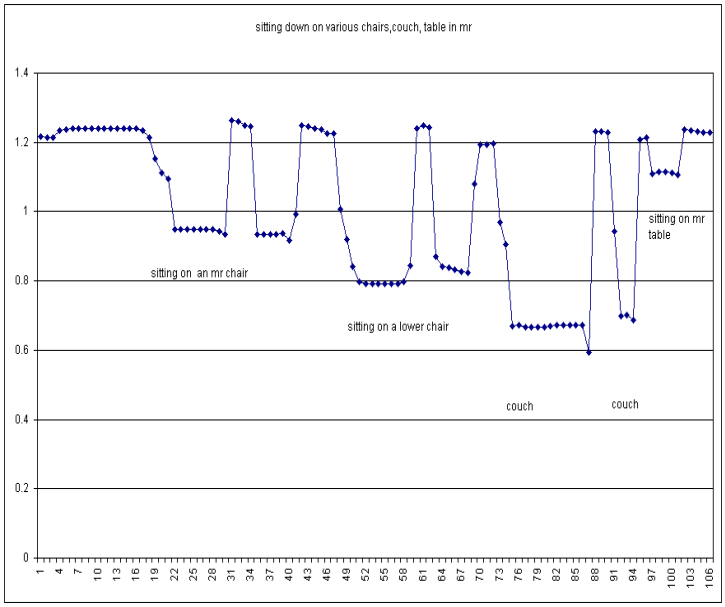
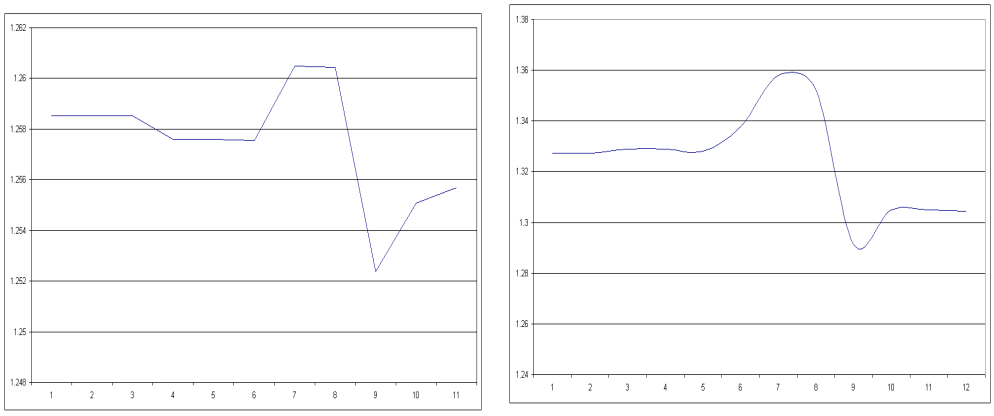


Figure 3.18. A sample of a user sitting on various chairs (z coordinate only)

Step Detection. The second approach, *step detection*, looks for turning points on the z axis that indicate a step. However this method does not behave well in the general case as standing still often has a similar pattern in the z coordinate as the walking sample, which is due to the error introduced in the measurements. The threshold method could be used in order to introduce a threshold in the size of the step that differentiates walking from being still, based on the size of the step. For example, Figure 3.19 portrays a *Still* and a *Walking* sample. These samples cannot be recognised by SPIRIT using step detection as the same problem as with height calibration arises again, i.e., there is no fixed threshold that can differentiate Walking samples from Still samples. Indeed, the displacement in Figure 3.19(b) is 0.07 cm vs. 0.008 cm displacement in Figure 3.19(a). If a threshold of 0.01 cm is set, where anything with a displacement above this value is a walking sample; the sample of Figure 3.7(b) that represents a Still sample portrays a height displacement of 0.03 that would classify it together with the walking sample of Figure 3.19. Thus, there is no obvious way of performing reliable recognition using step detection and thresholding (height calibration).



(a) (b)

Figure 3.19. A sample of a user remaining still (a) and walking (b) (z coordinate only)

3.9 Conclusions

- The most general conclusion is that HMMs are an appropriate methodology for creating middleware components as abstract models that represent movements such as the ones described in this chapter. The models that are created by HMMs are capable of recognition with high accuracy independently of the user and the specific topology of the implementation domain. They can be applied to different users and to different Sentient domains. Therefore, they are an appropriate methodology for the distributed nature of Sentient Computing.

- Recognition based on HMMs enable the creation of abstractions that were not recognisable before either due to a lack of an efficient method or due to noise. Also, the proposed recognition methodology resolves ambiguity in the *Sit Down* and *Stand Up* phonemes, which are directly dependent on the way a user carries a BAT and the specifics of the user and the physical environment.

- Because the set of recognisable phonemes is exclusive, the movement phonemes can be used in order to characterise each location sample with the movement it belongs to. This allows for "reminder events" to be created for each sample, which can then be used for communicating this information to remote nodes over unreliable protocols such as UDP. Reminder events are confirmations of the same abstract state, and therefore some loss can be tolerated.

- HMMs allow users to be recognised by their Sit Down and Stand Up phonemes. This can be quite useful in creating digital signatures, determining whether everybody has left the building in case of a security alert, etc. User recognition seems to work extremely well. The results advocate that the Sit Down phoneme is appropriate for such recognition, however, further experiments using a larger sample should be carried out in order to identify the system's behaviour when many users are of similar height. The influence of speed on the samples should be investigated as well. Section 3.11 describes a scenario where the Stand Up phoneme is used as a digital signature to ensure the successful evacuation of a building in case of an emergency. The success of the user recognition problem leaves scope for further recognition problems, including discovering the topology of the physical environment as is advocated by the Open Door Inwards phoneme.

- Lastly, a number of movements cannot be recognised by the Active BAT system, due to the displacement that is introduced in the measurement as well as the sampling rate being too low to capture enough information about the movement. The *spinning* movement is such a case and its track is portrayed in Figure 3.20.

3.10 Future Work

As future work, constraints can be built into the HMM network to identify legal sequences of movements. For example, a user cannot be seated and walking at the next instance. A Stand Up movement should be inferred. Specific-case recognition can be scheduled as a second layer after general-case recognition (see Figure 3.21). For example, if it is determined that the user has sat down, then specific case recognition can be performed on the samples that represented the sitting state and identify the object the user is sitting on. Or, when it is determined that the user is walking, a second recognition layer can take place which tries to infer doors being opened, thus indicating that the user has turned into a specific room.

Some *Sitting* samples are portrayed in Figures 3.22 and 3.23. Note that in the samples that correspond to sitting on the couch, the error in the y coordinate seems to be much smaller than the error in the x coordinate. This is believed to be due to the position of the couch that is placed against the wall and therefore it is visible to a smaller set of sensors than the chair that is placed in the middle of the room. This can be verified with further tests.

Finally, it is worth investigating whether movements that are not currently detectable by HMMs using data from the Active BAT, such as *spinning* and *directional walk*, can be sampled more frequently with alternative technologies of at least similar accuracy to the Active BAT.

3.11 Applications

On Demand Dynamic Memory Management. An area where the proposed recogniser can be directly applied is optimising the page-loading scheme in a location middleware system such as SPIRIT. By knowing the state of the user in terms of his movements, the system can make decisions as to whether the user should be monitored or not. In a system such as SPIRIT, which is object-oriented, users are represented by objects. If the system decides that the user should not be monitored for some interval, then the object that corresponds to that user is a good candidate for being replaced in the system cache. That can improve the performance of the middleware component.

Distribution. One of the main advantages of the proposed model is that it is independent of the underlying topology of the physical environment. This enables it to be trained once and be subsequently ported into any other environment that supports location services, as well. In fact, it can be used to discover the topology of the environment.

Digital Signatures. The Stand Up phoneme can be used as a digital signature after an alarm occurs in a building signalling an evacuation. By monitoring all users getting up and performing user recognition on all the samples, the system can determine if everybody has left the building or not.

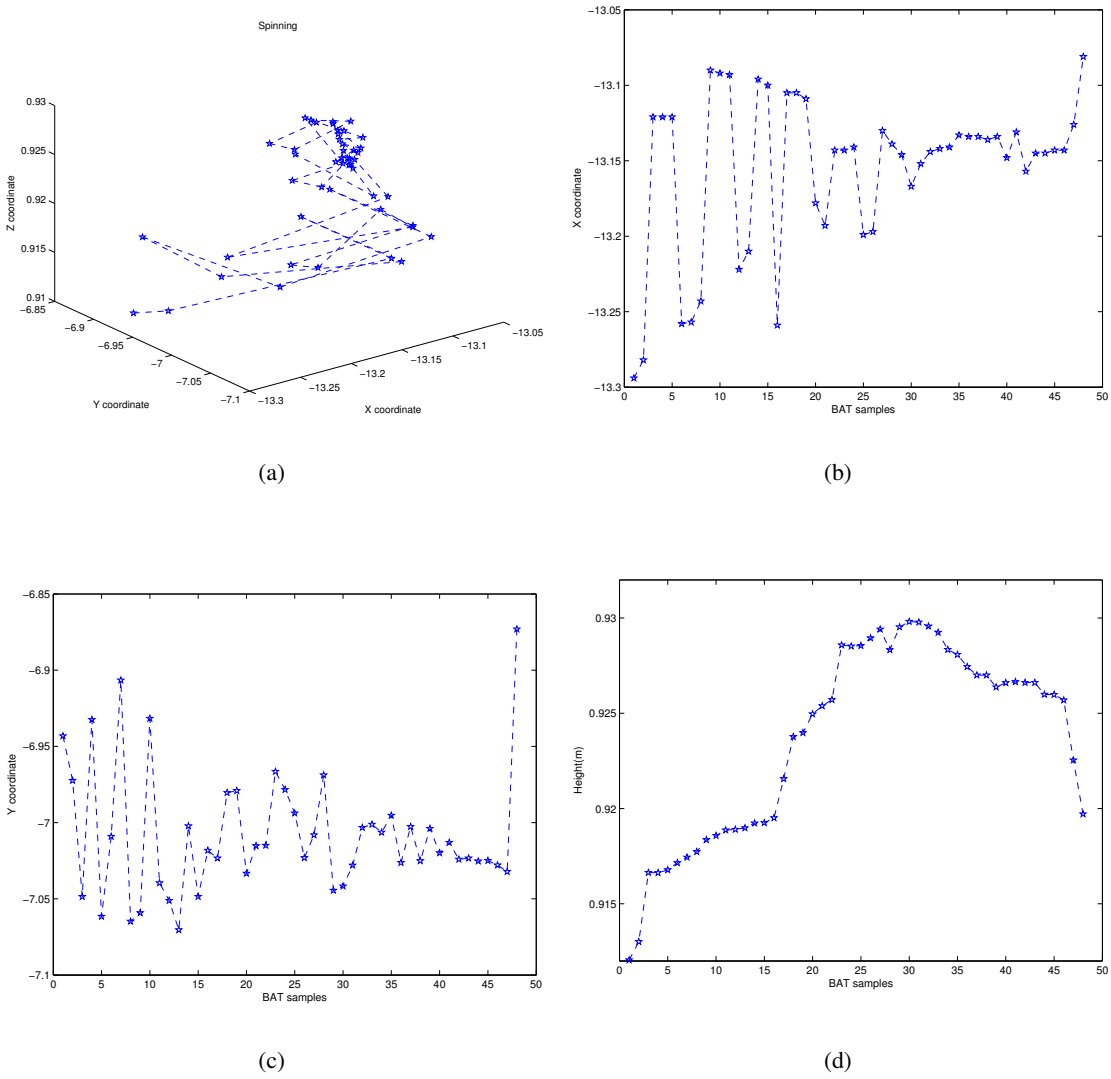


Figure 3.20. Spinning track (a) x coordinate (b) y coordinate (c) z coordinate (d).

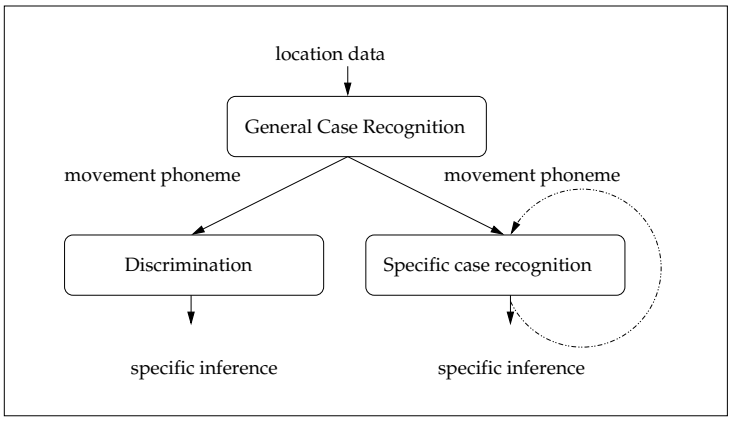


Figure 3.21. Layered recognition.

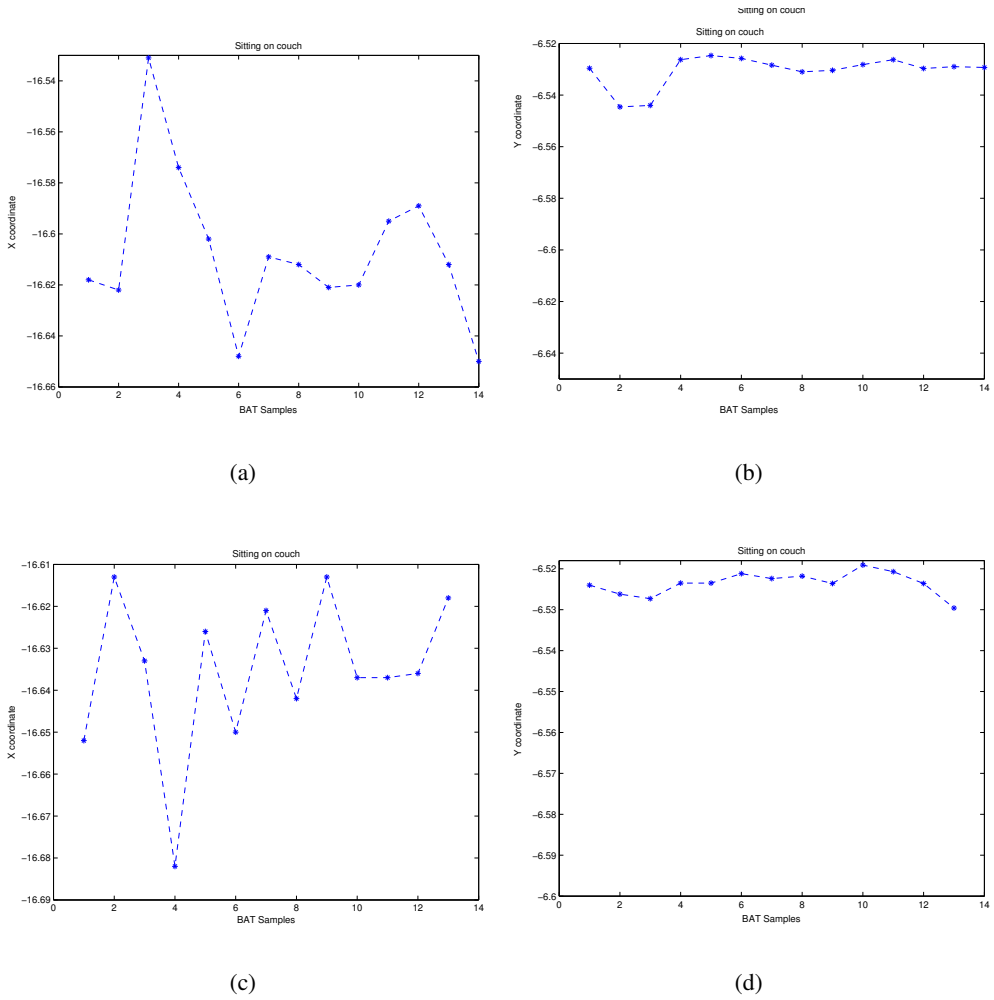


Figure 3.22. Two Sitting samples (LCE couch)

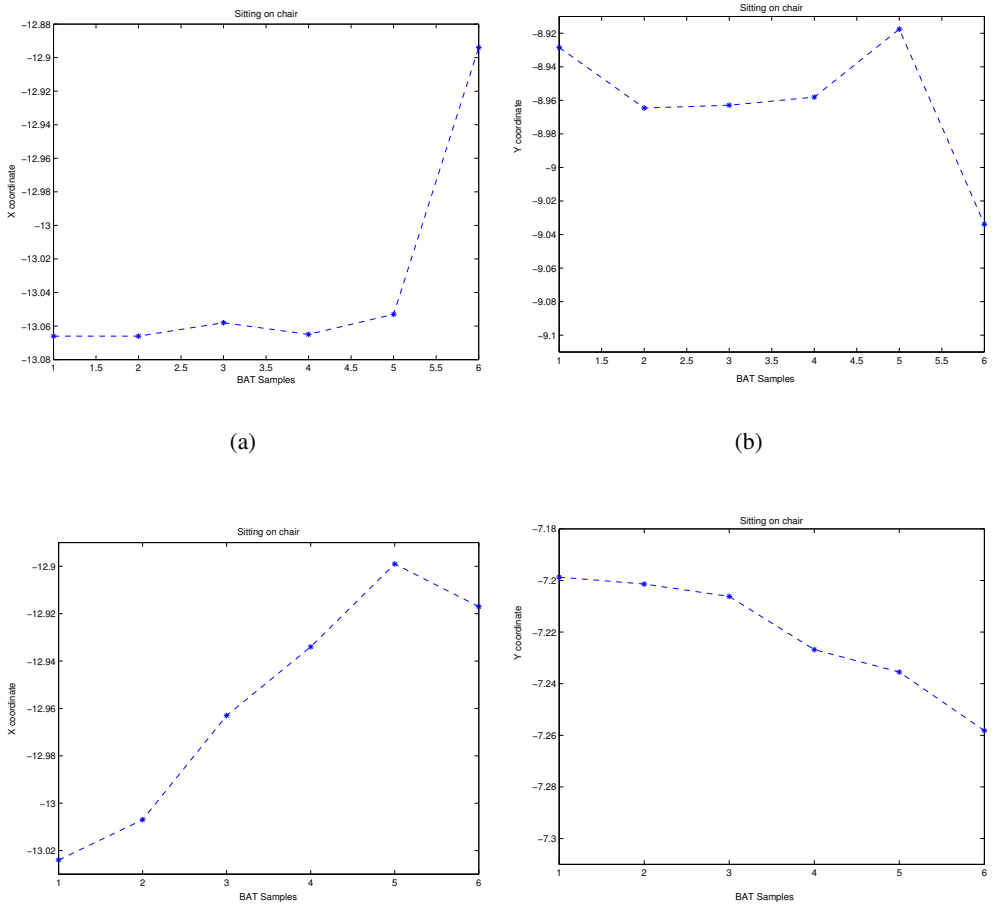


Figure 3.23. Two Sitting samples (LCE chair)

Chapter 4

Prediction

This chapter investigates the possibility of estimating the likelihood that a future instance of a knowledge predicate will be generated (either produced directly by the sensors or deduced from other predicates) in a knowledge base component in the sensor-driven system. More specifically, this work focuses on investigating the applicability of *Bayesian Prediction* for modelling the likelihood for holding knowledge predicates. These models form part of SCAFOS and are generic enough to be applicable to a large number of sensor-driven components. The results of this investigation are used in order to build a *prediction system*. Prediction is very important for Sentient Computing, as it increases the potential for *decision making*, even in cases where data sources are unavailable. Prediction also enables a *trade-off* between *certainty* and *cost*, as is demonstrated in this chapter.

4.1 Prediction

Chapter 3 focused on inferring abstract knowledge, such as user movements, from concrete predicates, such as user positions. This chapter discusses prediction as a methodology that is applicable to both concrete and abstract knowledge predicates (discussed in more detail in Chapter 5), such as

$$H\_UserInLocation(uid, rid, role, rattr, timestamp)$$

and aims to define a probability model that estimates the probability that a particular predicate instance will be generated in the system (in the knowledge base of a Deductive KB component in Figure 1.1) in the future. The methodology used for the prediction is based on the *naïve Bayes classifier* [67] seen as an equivalent Bayesian network [67]. Using the naïve Bayes classifier, a user's next location can be predicted from historical data. The same methodology can be applied to any knowledge predicate for which historical data is available. A prediction system such as the one postulated in this chapter provides an estimation of the likelihood that an instance of a knowledge predicate of interest will be generated in the future. The predicted instance can be derived directly from sensors, or deduced from other predicates. Such a system has the following benefits:

- It enables decision making about future knowledge. For example, a user may decide to create an application which will notify him when the coffee is ready only if the probability that somebody will make coffee in the next two hours is more than 50%.

- It can be used when the location system is unavailable or in order to reduce the monitoring cost. For example, using a prediction system it is possible for the system to return an answer such as "John's position is unavailable at the moment but possible locations are Room 10 and the Meeting Room" when John's position is queried and estimates are tolerated by the querying application.

4.1.1 The Naïve Bayes Classifier

The *naïve Bayes classifier* is a widely used practical learning method that can be used to address the problem of supervised learning. A set of training instances $x_1, \cdots, x_k$ is provided as well as a target function $f(x)$ which can take on any value from V. Each training instance x_i is a pair of a tuple of attribute values $\vec{a_i} = \langle a_{i1}, \cdots, a_{in} \rangle$ and the value of the target function for this instance $c = f(x_i)$ that acts as a *label* for that instance.

$$\begin{aligned} x_1 &= a_{11}, \cdots, a_{1n}, c_1 \\ x_2 &= a_{21}, \cdots, a_{2n}, c_2 \\ \vdots & \quad \vdots \\ x_k &= a_{k1}, \cdots, a_{kn}, c_k \end{aligned}$$

A new instance x_0 is presented, described by the tuple of attribute values $\langle a_1, a_2, \cdots, a_n \rangle$. A classifier needs to be constructed that can predict the target value $c_0 = f(x_0)$ for that instance. In order to classify the new instance, the naïve Bayes classifier assigns the target value with the *maximum a posteriori* (MAP) probability v_{MAP} given the attribute values $\langle a_1, a_2, \cdots, a_n \rangle$ that describe the new instance. The notation $\arg\max_{c_0} |\Theta(c_0)$ is used in order to denote the value of v_0 that maximises the term $\Theta(v_0)$.

$$v_{MAP} = \arg\max_{c_0 \in V} P(c_0 | a_1, a_2, \cdots, a_n) \tag{4.1}$$

Using Bayes theorem this expression becomes

$$\begin{aligned} v_{MAP} &= \arg\max_{c_0 \in V} \frac{P(a_1, a_2, \cdots, a_n | c_0) P(c_0)}{P(a_1, a_2, \cdots, a_n)} \\ &= \arg\max_{c_0 \in V} P(a_1, a_2, \cdots, a_n | c_0) P(c_0) \end{aligned} \tag{4.2}$$

Note that the denominator $P(a_1, a_2, \cdots, a_n)$ is dropped because it is a constant, independent of c_0. The two terms in (4.2) can be estimated from the training data. $P(c_0)$ is the relative frequency with which each target value c_0 occurs. The term $P(a_1, a_2, \cdots, a_n | c_0)$ can be estimated by counting the number of times with which each target value c_0 occurs in the subset of the training data for which $f(x) = c_0$. The naïve Bayes classifier is based on the simplifying assumption that the attribute values are conditionally independent given the target value. In other words, the assumption is that given the target value of the instance, the probability of observing the tuple $\langle a_1, a_2, \cdots, a_n \rangle$, namely $P(a_1, a_2, \cdots, a_n)$, is just the product of the probabilities for the individual attributes and so: $P(a_1, a_2, \cdots, a_n | c_0) = \prod_i P(a_i | c_0)$. Substituting this into (4.2), we have the approach used by the naïve Bayes classifier, where v_{NB} denotes the target value output by the naïve Bayes classifier.

$$v_{NB} = \arg\max_{c_0 \in V} P(c_0) \prod_i P(a_i | c_0) \tag{4.3}$$

4.1.2 Bayesian Networks

Recall the notation and nomenclature used in *Directed Acyclic Graphs (DAGs)*. A DAG is a pair (X, E), where $X = \{X_1, \cdots, X_{n+1}\}$ is a set of vertices, and E is a set of edges between the vertices. The graph is acyclic, i.e., no path starts and ends at the same vertex. The graph is also directed.

Using statistical nomenclature, $U_1, \cdots, U_{n+1}$ are *random variables*. A *Bayesian Network* is an efficient representation of the *joint probability distribution* over a set U. The joint probability distribution

	Alastair	*Andy*	*Alan*
Room 1	0.6	0.3	0.2
Room 2	0.3	0.3	0.3
Room 3	0.1	0.4	0.5

Table 4.1. Conditional probability table for $P(Rid|Uid)$

specifies the probability for each of the possible variable bindings for the tuple $U = \langle U_1, \cdots, U_{n+1} \rangle$. Formally, a Bayesian network for U is a directed acyclic graph G whose nodes X correspond to the random variables in U and whose edges represent direct dependencies between the variables according to the following dependency assumptions: each node X_i is independent of its non-descendants, given its parents in G. X is a *descendant* of Y if there is a directed path from Y to X. The set of the immediate predecessor (parent) nodes of X_i is denoted by the term $Parents(X_i)$. A conditional probability table is given for each variable X_i describing the probability distribution for that variable given its parents. The joint probability for any desired assignment of values $\langle x_1, \cdots, x_{n+1} \rangle$ to the tuple of network variables $\langle X_1, \cdots, X_{n+1} \rangle$ can be computed by the formula:

$$P(x_1, \cdots, x_n) = \prod_{i=1}^{n} P(x_i|Parents(X_i)) \tag{4.4}$$

The values of $P(x_i|Parents(X_i))$ are stored in a conditional probability table associated with node X_i.

Example. To illustrate, consider a node $X = Rid$ which is dependent on the node $Y = Uid$. The node $X = Rid$ represents the names of the rooms in an office and takes a value from the set of possible values *Room 1, Room 2, Room 3*. The node $Y = Uid$ represents the names of the people that work in that office and can take any value from the set *Alastair, Andy, Alan* . The conditional probability table for Rid given Uid reflects the probability that the location of a user in *Alastair, Andy, Alan* will be one of the values of the set *Room 1, Room 2, Room 3* and is given in Table 4.1. For example, the conditional probability that an instance of the predicate $H\_UserInLocation(uid, rid)$[1] where uid=*Alastair* will contain the constant *Room 1* as the value of rid is 0.6. That is interpreted as follows: The probability that Alastair will be located in *Room 1* is 0.6.

4.1.3 Bayesian Networks that Correspond to the Naïve Bayes Classifier

The naïve Bayes classifier can be equivalently viewed as a simple Bayesian network of the structure depicted in Figure 4.1. In this network, every leaf is an attribute of the classification and it is independent from the rest of the attributes, given the state of the class variable, namely the root of the network. Each node therefore has only one parent, the root node. The equivalence of the Bayes classifier and the network of Figure 4.1 can be drawn from Equation (4.4) as follows: Let $U = \{A_1, \cdots, A_n, C\}$, where the random variables $A_1, \cdots, A_n$ are the attributes and the random variable C is the class variable. In the network of Figure 4.1 the class variable is the root, i.e., $Parents(C) = \{\}$, and the only parent for each attribute is the class variable, i.e, $Parents(A_i) = \{C\}$, for all $1 \leq i \leq n$. Using (4.4), it can be deduced that $P(A_1, \cdots, A_n, C) = P(C) \prod_{i=1}^{n} P(A_i|C)$. The value that maximises $P(A_1, \cdots, A_n, C)$ is given by naïve Bayes definition (Equation (4.1)).

[1]For simplicity reasons the values *role* and *rattr* are ignored in this example.

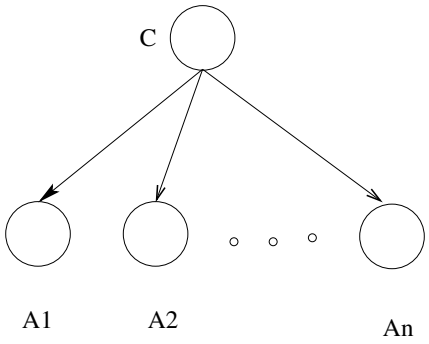


Figure 4.1. Bayesian network for the naïve Bayes classifier.

4.1.4 Using the Naïve Bayes Classifier to Predict Knowledge Predicates

Considering the predicate

$$H\_UserInLocation(uid, rid, role, rattr, timetamp),$$

any of the variables uid, rid, $role$, $rattr$, $timestamp$ can be seen as the class variable, and the rest of the attributes can be seen as the attribute variables, according to which the most probable value for the class variable can be computed, given a new instance. The Bayesian network of Figure 4.2 represents the naïve Bayes classifier for the predicate $H\_UserInLocation(uid, role, rid, rattr, timestamp)$ where the class variable is *uid* and the attribute variables are *role,rattr,timestamp*, according to equation:

$$uid_{MAP} = \underset{uid \in Users}{\arg\max}\, P(uid, role, rid, rattr, timestamp) \tag{4.5}$$

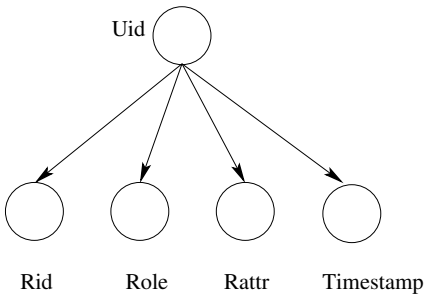


Figure 4.2. A Bayesian network for classifying uid according to rid,$role$,$rattr$,$timestamp$ for the $H\_UserInLocation$ predicate.

Using this network, the joint probability distribution for the variables $uid, rid, role, rattr, timestamp$ can be calculated by using the following equation:

$$P(uid, role, rid, rattr, timestamp) = P(uid)P(role|uid)P(rid|uid)P(rattr|uid)P(timestamp|uid) \tag{4.6}$$

The probability density for class Uid i.e., the probability of all values of the attribute uid that correspond to users who are PhD students and are located inside *Room 10*, which is an office, between 3 am and

9 am in the morning can be calculated from Equation 4.6:

$$P(uid, Phd, \textit{Room 10}, \textit{Office}, \textit{3-9}) = \\ P(uid)P(Room10|uid)P(\textit{Office}|uid)P(Phd|uid)P(Timestamp|uid) \tag{4.7}$$

The most probable Phd student to be found in Room-10 between 3 and 9 is the one for whose identifier uid, the term $P(uid, Phd, \textit{Room 10}, \textit{Office}, \textit{3-9})$ has the maximum value.

$$uid_{MAP} = \underset{uid \in Users}{\arg\max}\ P(uid, Phd, \textit{Room 10}, \textit{Office}, \textit{3-9}) \tag{4.8}$$

Note that *Users* is a finite set of user identifiers (Chapter 5). In order to predict a user's location at a given point in time, a different Bayesian network needs to be employed, that of Figure 4.3 where the class variable is rid instead of uid. This network can be used in order to calculate the value rid_{MAP}, given a new instance $H\_UserInLocation(John, Phd, rid, rattr, timestamp)$.

$$\begin{aligned} rid_{MAP} &= \underset{rid \in Regions}{\arg\max}\ P(rid)P(John|rid)P(PhD|rid)P(rattr|rid) \\ &= \underset{rid \in Regions}{\arg\max}\ P(rid)P(John|rid)P(PhD|rid) \end{aligned} \tag{4.9}$$

This is equivalent to calculating the most probable location where John will be sighted by the location system. $Regions$ is the finite set of all regions (rooms) known to the system (Chapter 5). Similarly, the predicate

$$H\_ClosestEmptyLocation(uid, Sysadmin, rid, Kitchen, timestamp),$$

can be used to predict the most probable value that identifies the closest, empty location of type $rattr = Kitchen$ with respect to any user with the property $role = Sysadmin$:

$$rid_{MAP} = \underset{rid \in Regions}{\arg\max}\ P(rid)P(Sysadmin|rid)P(uid|rid)P(Kitchen|rid)P(timestamp|rid)) \tag{4.10}$$

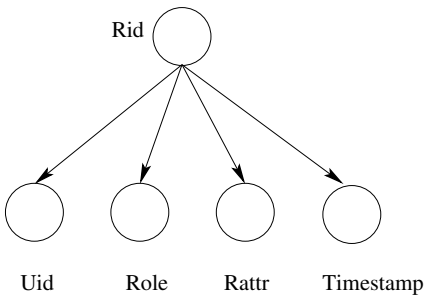


Figure 4.3. A Bayesian network for classifying rid according to uid,$role$,$rattr$,$timestamp$ for the predicate $H\_UserInLocation$.

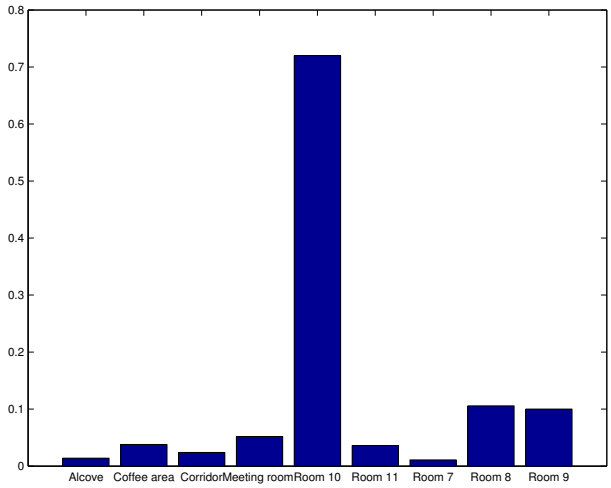


Figure 4.4. Probability density of Mike's locations from 3 am to 11 pm.

4.1.5 Prototype Implementation

A set of experiments were carried out in the LCE using the Bayesian networks of Figures 4.2 and 4.3. The networks were trained using of a set of location data (instances of the $H\_UserInLocation$ predicate), which was produced by monitoring the movements of nine users (*Alastair, Professor A (Andy), Dave, David, James, Jamie, Kieran, Mike, Robert*) by means of the Active BAT location system [41]. The monitoring was restricted to the top floor of the LCE, which consists of the following locations: *Alcove, Coffee area, Corridor, Meeting room, Room 10, Room 11, Room 9*. An analysis of all the sightings of the above users over a period of 72 hours using an implementation of the naïve Bayes classifier as a Bayesian network (B-Course tool [69]) produced the results that are discussed here. For each experiment the outcome of the classifier is the probability density of the class variable. The size of the training set is 22,280 instances (location sightings).

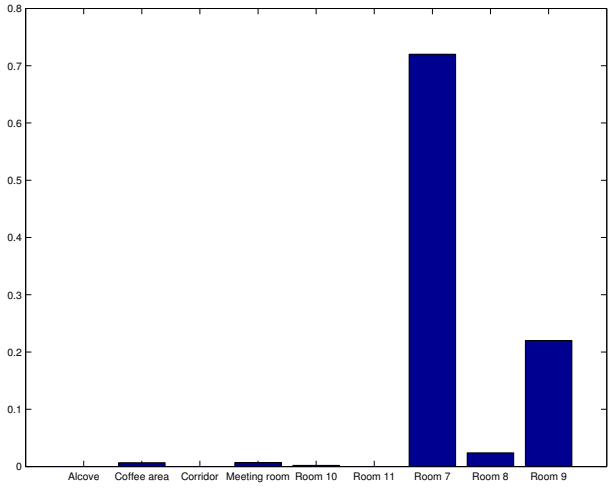


Figure 4.5. Probability density of Professor A's locations (3am–9am)

4.1.6 Experiments

Experiment 1. *Where is Mike likely to be seen?*

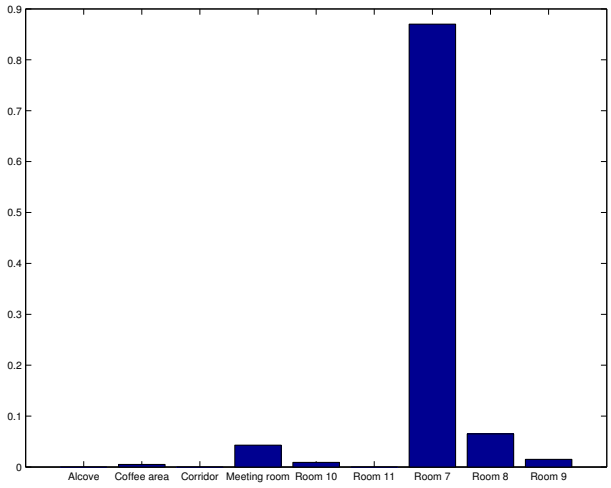


Figure 4.6. Probability density of Professor A's locations (10am–4pm)

Figure 4.4 portrays the probability density for the LCE locations where Mike may be seen, based on his past movements, as captured by the training data. Room 10, which is Mike's office, appears as the most probable location. This is the expected result.

Experiment 2. *When can I meet Professor A ?*

In order for Professor A's assistant to arrange appointments with undergraduate students, Figures 4.5, 4.6 and 4.7 suggest the probability density of the LCE locations where Professor A can be seen during the day, based on his past movements. The most probable location is Room 7, which is Professor A's office. Rooms 7, 8 and 9 are adjacent and Professor A often has informal meetings in these adjacent rooms; this is clearly visible in Figures 4.5, 4.6 and 4.7, as Rooms 8 and 9 appear to have higher probability than the rest. The meeting room is also used by Professor A during the day, as is verified by the same figures.

Experiment 3. *Where shall I look for David first?*

This is an example of calculating the maximum likelihood for a user's location. Figure 4.8 shows that David's sightings are classifiable to Room 10 with the highest probability. This is an expected result, as David works in Room 10. David also spends time in Room 9 as he collaborates with someone who works there. This is clearly visible in Figure 4.8.

Experiment 4. *Which is the most probable location irrespective of user?*

The probability distribution for all LCE locations, irrespective of user, for the period between 3 am and 9 am, 10 am and 4 pm, and 5 pm to 11 pm is portrayed in Figures 4.9, 4.10 and 4.11, respectively.

Experiment 5. *Who is the most likely person to be in the meeting room between 10 am and 4 pm?*

The most probable person to be in the meeting room is the one whose who has been classified with the highest probability to the class that corresponds to the meeting room in Figure 4.12.

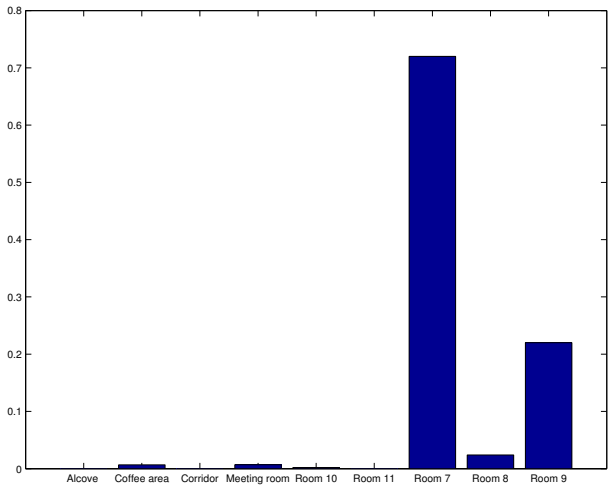


Figure 4.7. Probability density of Professor A's locations (5pm–11pm)

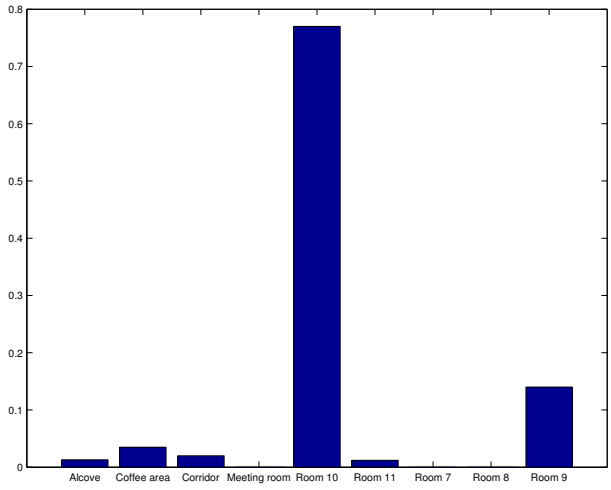


Figure 4.8. Probability density of David's locations at any time.

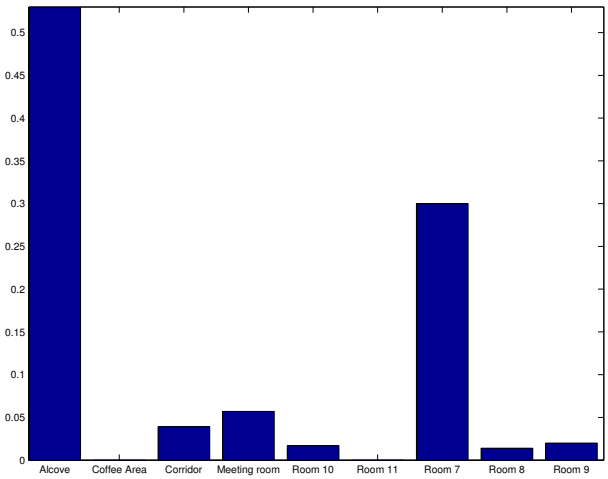


Figure 4.9. Location probability density irrespective of user between 3 am and 9 am.

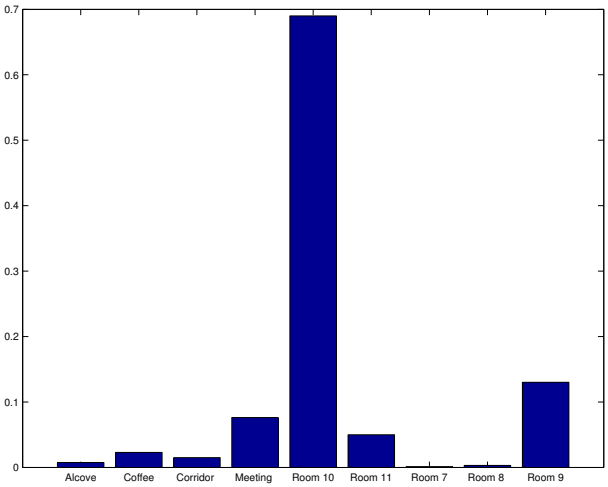


Figure 4.10. Location probability density irrespective of user between 10 am and 4 pm.

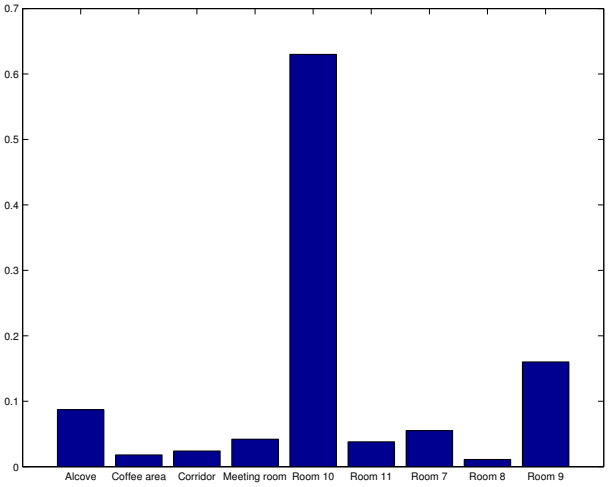


Figure 4.11. Location probability density irrespective of user between 5 pm and 11 pm.

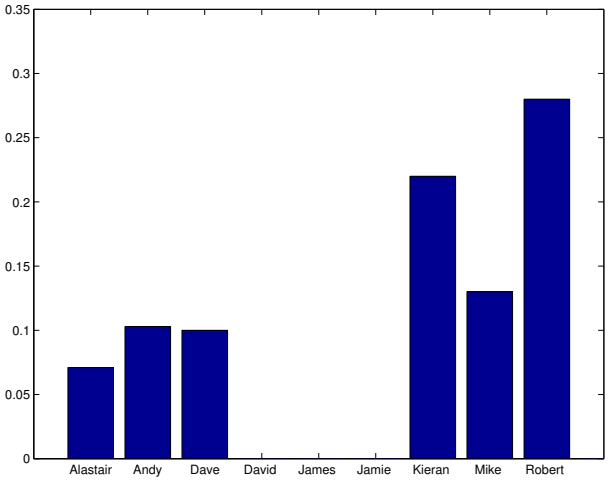


Figure 4.12. User probability density in the meeting room between 10 am and 4 pm.

4.2 Confidence Levels

A Bayesian network which is used for classification, such as the ones portrayed in Figures 4.2 and 4.3, can be associated with a measure of how successful the classification has been. This measure is called the *classification success score* or *classification accuracy*, or *prediction accuracy*, and is calculated using the *Leave-one-out cross-validation* method [67].

Leave-one-out cross-validation is a method for estimating the predictive accuracy of the classifier. In this method, out of N training instances of the data set, $N-1$ instances are used in order to train the classifier and the remaining instance is used as a test instance for which the value of the class variable is to be calculated. This is repeated N times, each time, removing a different instance from the data set and using the rest $N-1$ instances as training data. The overall classification success score is calculated as the percentage of correct classifications over the overall classification attempts and it amounts to 55.24% for the network of Figure 4.2 and to 82.50% for the network of Figure 4.3.

The classification success score for each class is calculated as the percentage of correct classifications for that class. The reliability of the classification success score can be rated by the percentage of the training data that represents that class. The *classification success score* and the *reliability of the estimate* in terms of absolute sizes for the network of Figures 4.2 and 4.3 is shown in Figures 4.13 and 4.14, respectively. Both figures show that the classification success score depends on the size of the sample that corresponds to that class. The larger the size of the class, the larger the classification success score for that class. In Figure 4.13 the class *Andy* is more easily classifiable than *Alastair*, although the sample sizes for both classes are similar. This suggests that Alastair's sightings are dispersed into more rooms than Andy's sightings.

The classification success score and the reliability estimate can be combined in order to form a *confidence level* for evaluating the predictive classification. The combined result, consisting of the classification success score and the confidence level, is used in order to characterise the predicates of the model of Chapter 5.

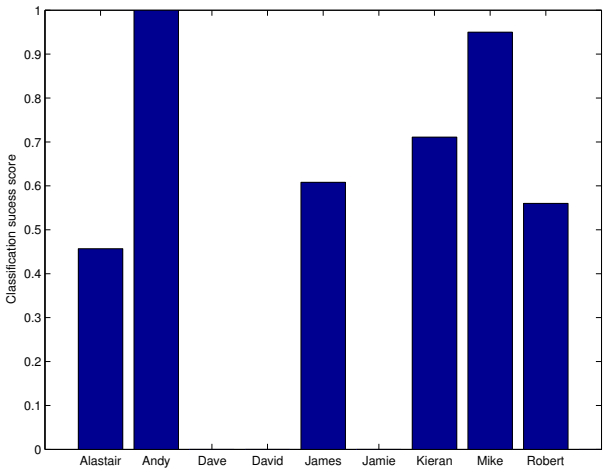
$$Inference\ \textit{Confidence Level} \quad = \quad \{classification\ success\ score, reliability\ estimate\}. \quad (4.11)$$

4.2.1 Evaluating Rule-based Inference through Prediction

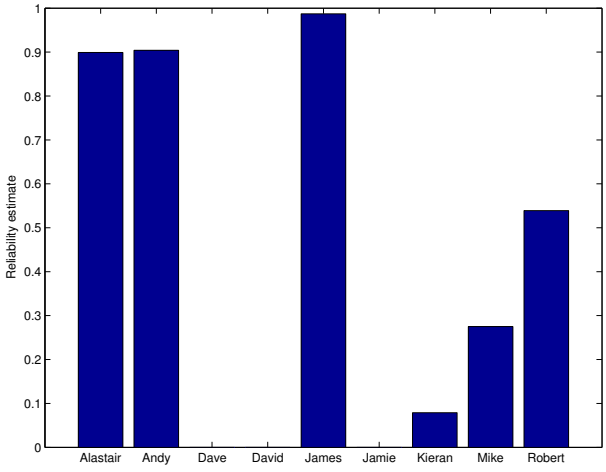
Likelihood estimation can be used in order to evaluate other forms of inferencing for the same predicate instance, such as the one based on logical deduction (Chapter 6). For example, a rule-based inference can be built based on the following assumption: in order to make coffee in the Laboratory for Communication Engineering, one must perform the following steps:

- Approach the coffee machine and remove the jar, which is either empty or contains the dregs of the previous coffee-batch.
- Approach the sink, empty the jar of its contents and fill it with fresh water from the tap.
- Re-approach the coffee machine, fill the coffee machine with the water, grind coffee, fill the coffee machine with coffee and press the start button.
- Perform these steps within 2 minutes and without leaving the coffee-area in between.

The outcome of this experiment was four inferences, all of them correct. Three people were detected to be making coffee a total 4 times in 72 hours: Mike, Alastair and Eli. To evaluate this result, the locations of various LCE users in the coffee-area were analysed by means of Bayesian reasoning. The most probable people to be in the coffee-area in the morning are David, Mike and Alastair. Two of these people are regular coffee makers, as can be inferred by the predictability of their movements in the coffee-area in the morning, when coffee is usually prepared. David, although regularly seen in the coffee

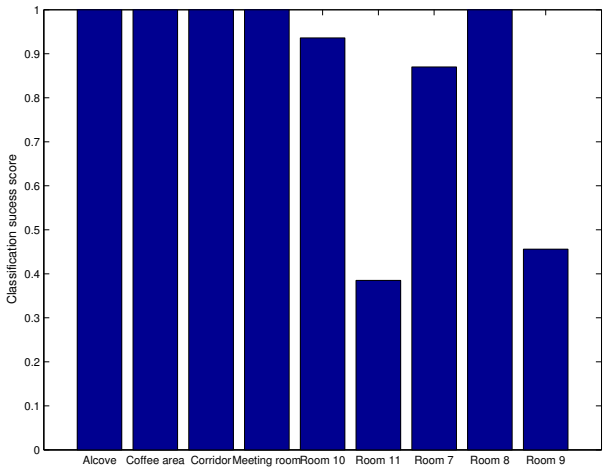


(a)

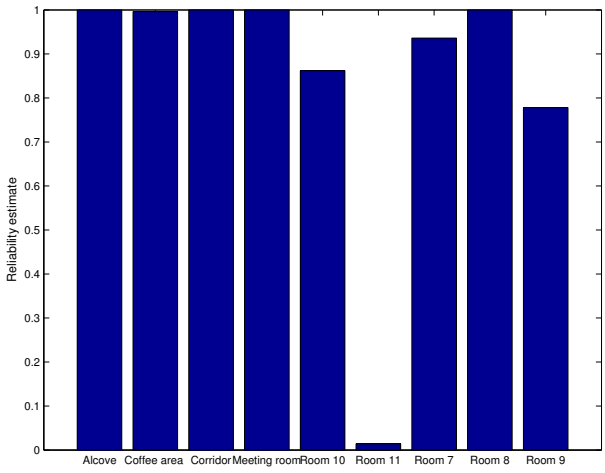


(b)

Figure 4.13. The classification score and the reliability estimate for the network of Figure 4.2.



(a)



(b)

Figure 4.14. The classification score and the reliability estimate for the network of Figure 4.3.

area in the morning, does not drink coffee and was, therefore, making tea. Eli made coffee once during the experiment, too.

4.2.2 Discrete vs. Continuous Variables

The instances to be predicted can be discrete by nature, e.g., containment within a room, or of a continuous nature, e.g., position streams and timestamp sequences; in this case, they need to be appropriately discretised. In the implementation described in this chapter, the $timestamp$ variable was discretised into fixed length intervals. The choice of these intervals is not optimal, as it would be more useful to consider the intervals 11pm-7am (night) 7am-1pm (morning) and 1pm-11pm (afternoon-evening). Discretisation can also be achieved by means of the k-means clustering algorithm [4] or the nearest neighbour algorithm [55].

4.2.3 Network Optimisation

The networks of Figures 4.2 and 4.3 contain all the classification variables. However, some of the variables are more important than others, and choosing only a subset of important variables can improve the classification accuracy. For example, when classifying the variable uid according to the features $rid, rattr, role, timestamp$ it is better to omit the attribute $rattr$ from the classification, as each region with identifier rid may have more than one regional attribute and the same regional attribute (e.g., *Office*) can characterise more than one location. The network structure can therefore be optimised by selecting the subset of attributes that gives the best classification accuracy.

Several algorithms exist for selecting the subset of variables that produces the best classification success score. *Greedy* search is an algorithm in which models are created for all possible subsets of attribute variables and their classification accuracies are compared one by one, until all models have been exhausted. This method is clearly inefficient when the number of models is large. Other solutions include searching using the *Hill Climbing* algorithm, the *tabu search* algorithm or even *genetic algorithms*.

The network of Figure 4.2 has an overall classification success score of 55.24%. When optimised by removing the variable $rattr$ its classification accuracy becomes 56.20%. Similarly, the overall classification accuracy of the network of Figure 4.3 is improved from 82.50% to 85.19% when the attribute $role$ is removed from the attribute set.

4.3 Conclusions

The results indicate that the naïve Bayes classifier is an appropriate method for predicting the likelihood that a future predicate instance will be generated in the system. The naïve Bayes classifier is powerful, and, although it assumes that all the predictor variables are conditionally independent, it has impressive results even for the cases where there are dependencies between the variables, such as the pairs *role-uid* and *rid-rattr*.

The classification accuracy is a measure of the predictive power of the classifier. The reliability of the likelihood estimation depends on the size of each class, which is a percentage of the size of the test set. Although prediction was demonstrated for a location ($H\_UserInLocation$) predicate, it can be re-applied to other high-level knowledge predicates too, e.g., $H\_ClosestEmptyLocation$, $H\_UserCoLocation$.

Bayesian prediction can be used in SCAFOS in order to enable decision making, even when data sources that provide the data for the decision making are unavailable, e.g., in case the location system fails. The estimated likelihood for a predicate instance can be used to form application specifications using SCALA (Chapter 12), thus enhancing expressiveness. For example, an application may be interested in knowing the probability that meetings between at least 20 people will take place during the day, and if the probability is more than 50%, a new batch order for cookies should be issued. The user may query

the probability of an event in order to decide to respond differently to a very likely situation rather than an unlikely one; e.g., he may decide not to register for notifications for an unlikely situation. In general, prediction is very important for Sentient Computing because it constitutes an acceptable solution when trading information certainty for resource savings (such as computational cost and a user's time), or when certainty is compromised by a failure. For example, instead of a user's position, an estimation of where the user may be is given as the answer to a query (Chapter 5) or published as an event (Chapter 8). This does not require the location system to monitor that user, thus saving on computation and communication cost. A quantification scheme, where a user's time is treated as a resource that is associated with a cost, is discussed in [100].

As future work, it will be interesting to look at other models that take dependencies into account, such as full Bayesian networks, and compare their performance to the naïve Bayes classifier.

Chapter 5

A Conceptual Framework

Chapter 1 discusses the importance of context-awareness. Chapter 2 discusses sensor-driven systems and context aware applications in an attempt to understand what features are common and useful across context-aware applications in such systems. This chapter explains why traditional event-based models are not appropriate for sensor-driven systems and presents a conceptual framework that provides the required features for building applications, as well as a programming model that makes application development easier.

5.1 Requirements of Context-Awareness in Sentient Computing.

As mentioned earlier, Sentient Computing systems monitor the stimuli provided by environmental sensors in order to *notify* applications of changes that are relevant to the changing context of the user; for example, his identity, location or current activity. Sentient Computing applications follow a common modus operandum: the user specifies his requirements in terms of abstract, high-level context as well as the service to be delivered, once the specified context has occurred. The application receives notifications from the Sentient system whenever the specified context happens, and it executes accordingly the specified action. For example, the user may ask to be reminded to return John's book, when he is in the same room as John.

Current trends in programming paradigms in the area of context-awareness, such as *ubiquitous* and *pervasive computing*, advocate a separation of concerns between the way the user perceives knowledge about the physical world and the way knowledge is produced and maintained within Sentient Computing systems. The user's view is *abstract and state-based*, i.e., events are perceived as changes of state. The user's view should be *transparent*, i.e., the required application functionality should be available irrespectively of the heterogeneity of the underlying distributed modelling (in terms of the physical entities it contains) and the heterogeneity should be concealed, even when the user is mobile. For example, locating the *closest, empty, meeting room* should be feasible, both when the user is walking in the Computer Laboratory and within the LCE. Furthermore, to accommodate user requirements, Sentient Computing environments need to be *dynamically extensible in real-time*. New user requirements need to be satisfiable without taking the system offline or recompiling existing applications, even when entities are added or removed from the underlying model.

Summarising the above, the following requirements have been identified for modelling context-awareness:

1. Transparency in reasoning with distributed state in heterogeneous sensor-driven components.
2. Dynamic extensibility.
3. State-based, integrated knowledge modelling.

4. Separation of concerns between knowledge management and knowledge usage in applications.

5.1.1 Chapter Layout

This chapter identifies the deficiencies of current event-based knowledge for satisfying the above requirements and proposes an alternative model. First, it defines *transparency* for context-aware sensor-driven systems. Next, it compares state-based to event-based modelling approaches and demonstrates that state-based modelling satisfies the above requirements while event-based modelling does not. The final part of this chapter presents a state-based model for context-awareness in Sentient Computing that is compatible with the above requirements.

5.2 Transparency

We extend the ODP [48] definition by defining *Context-Aware Application Transparency (CAAT)* to signify the concealment from the user of the differences in the models of sensor-driven distributed components. CAAT is very important in sensor-driven systems because it advocates that specifications of user requirements, in terms of knowledge (see Section 5.1), are compiled into a generic implementation which is, in turn, irrespective of the underlying sensor-driven system. Without CAAT, the same subscription specification would have to be implemented differently in each publisher domain.

Dynamic Model Extensibility. Transparency is crucial to dynamic extensibility, i.e., the extension of the sensor-driven model by new entities such as regions or users. In fact, when a new region or a new user is added to the model, the mapping between the language expression (that is implemented by the FSMs) and the domain changes, rendering the existing implementation inadequate (see Appendix A) and requiring re-compilation. Transparency, in this case, advocates that the implementation need not be recompiled after the sensor-driven model is modified in such a way.

Application-driven Dynamic Extensibility. Application-driven dynamic extensibility refers to the extension of the model with new abstract knowledge predicates, as specified by the user. Transparency is crucial in this case, as well, as each new abstract predicate needs to be abstracted from existing knowledge in the same way, in each implementation domain of each distributed component. The reasoning mechanism for abstracting knowledge from a systems' perspective is described in Chapter 6 and from a distributed systems' perspective in Chapter 8.

Context-Aware Application Roaming. Another case when the heterogeneity of each domain needs to be masked is the case of follow-me applications and mobile agents. A follow-me application needs to work as the user moves from domain to domain. For example, a follow-me assistant may be interested in locating the closest empty room to the user it belongs to, as that user moves from one domain to another. We refer to this particular case of transparency as *Context-Aware Application Roaming* (CAAR).

5.3 State-Based vs. Event-Based Modelling

A state based-approach is a novel way of looking at sensor-driven systems, since all approaches so far have been event-based. In many systems, event-based and state-based modelling have been considered as equivalent approaches. State-based modelling for context-awareness in sensor-driven systems satisfies the requirements of Section 5.1, while, as is demonstrated in this chapter, event-based approaches violate *dynamic extensibility*, *semantic transparency* and *separation of concerns between knowledge management and knowledge use*.

5.4 State-Based Modelling

State-based modelling for sensor-driven systems means that the state of an entity is modelled by means of a set of logical predicates (state predicates). State is extracted from each event and is made persistent in memory. If the next event that arrives contains the same state, it is ignored. If not, state for that entity is updated and a higher-order event is fired.

This is appropriate for sensor-driven systems for the following reasons:

- Events in sensor-driven systems are not always characteristic or semantically important. Instead, they can be seen as snapshots of the state of a physical entity at a given point in time. For example, events are produced by a location system according to its sampling rate, even when the user remains idle. This is different from, say, a game of squash where each event is modelled to signify the point where the ball hits the wall, which is the basis by which the score is calculated.
- Events in sensor-driven systems do not convey the fact that something didn't happen or no longer happens. This needs to be deduced by the system based on the received events or lack of them.
- State-based representations are more natural for the user than event-based representations. For example, the phrase "notify me when user A enters the meeting room and then user B enters the meeting room without user A having left the meeting room first" is intuitively more complex than the equivalent state expression "notify me when users A and B are in the same room".
- A state representation is, therefore, compatible with the statistical inferencing methodologies of Chapter 3. The latter produces, by default, descriptions of object state.
- A state-based modelling approach is more appropriate for distributed systems which are characterised by unreliable communications such as packet loss and node failure. In the case of node failures, the state is stored and can be re-instated. In the case of packet loss, reminder events can be added so that transmission can be done over unreliable transport protocols, such as UDP.

5.5 Event-Based Modelling

In event-based modelling, the state of an entity is represented by means of an *event history* entity. Having an event-based approach to modelling sensor-driven systems means that each publisher in a sensor-driven network sends on to the subscribers all the generated events from the sensors of a particular context through a *notification service*. More sophisticated solutions exist for *filtering* notifications and creating *event compositions* [6, 7, 63, 81, 102].

5.5.1 Deficiencies of Current Event Models

- As these approaches are implemented with FSMs, no persistent storage is normally available. This is a problem when a node is restarted; in such a case, *uncertainty* and *partial knowledge* is introduced. This is discussed in detail in Section 5.6.
- Because primitive events correspond to samples of user tracks and users move freely around, a state such as "an empty room" can be reached in many random ways. Using an event model, a subscription to a composite event needs to be created for each possible way in which a meeting room can become empty.
- Another issue arises from the fact that a user may decide to dynamically change a sensor-driven system model by adding a new region to be monitored or a new user. Current event models based on parametric finite state machines do not offer this possibility as a new finite state machine would

need to be created in place of the old one for implementing the same phrase. This is discussed in detail in Section 5.6.3.

- FSMs are dependent on the implementation domain and therefore cannot reason *transparently* with state in a distributed environment. This means that user requirements that define knowledge of interest cannot be mapped to a single, generic implementation that applies to each implementation domain in each component.

5.6 Deficiencies of Finite State Machines in Terms of Reasoning with Abstract State.

This section looks at FSM-based implementations of TFOL expressions with state predicates. We demonstrate that finite state machines have many shortcomings when reasoning with state that are due to three main underlying reasons: the first is that finite automata *cannot deal with negated predicates when these do not exist in the system*, which is an inherent process in reasoning with state. The second reason is that finite automata *lack memory structures* that allow them to store information about their alphabet, which introduces *partial knowledge*. The third reason is that the structure of an FSM is *dependent on the domain of implementation*. On the other hand, transparent reasoning with state requires both reasoning with negative (missing) abstract knowledge as well as concealing the details of the universe of discourse from the reasoning tool. Both issues are discussed in detail in the following section.

5.6.1 Implementing FOL Expressions with Negated Free Variables that Imply Lack of Information

In this category belong FOL phrases that signify lack of information, such as the ones that contain the operators$\neg$ and $\nexists$. Such expressions are not directly computable by a FSM. In fact, whenever negation occurs in an FOL expression signifying that the semantic mapping between the domain and the language is incomplete or missing, finite state machines cannot be used. As an example, the expression *"User A is nowhere"* is not satisfiable by a finite state machine (see Section B.2). This is due to an inherent inability of FSMs to deal with negation when this signifies lack of information.

There doesn't exist a finite-state machine which can accept the expression $\neg P$, when there is no instance of the predicate $\neg P$ stored in the knowledge base of the domain as a fact.

This means that a symbol that corresponds to either P or $\neg P$ must be explicitly generated for an FSM to be able to process it. However, in sensor-driven systems negative information is *absent by default.* Sensors only produce positive knowledge. The information that is produced by the sensors is in the form of a tuple:

$$\langle entity, state, timestamp \rangle$$

For example, monitoring user u_1 produces the following time-series:

$$\begin{aligned} &H\_UserInLocation(U_1, Room\ 1, T_1) \\ &H\_UserInLocation(U_1, Room\ 1, T_2) \\ &H\_UserInLocation(U_1, Room\ 2, T_3) \end{aligned} \tag{5.1}$$

The knowledge that must be deduced from that is:

$$H\_UserInLocation(U_1, Room\ 1, T_1)$$
$$H\_UserInLocation(U_1, Room\ 1, T_2)$$
$$\neg H\_UserInLocation(U_1, Room\ 1, T_3) \quad (5.2)$$
$$H\_UserInLocation(U_1, Room\ 2, T_3) \quad (5.3)$$

Assuming that user U_1 was co-located with user U_2 when he was in $Room\ 1$, and that user U_2 remained in $Room\ 1$ when U_1 moved to $Room\ 2$, then the following knowledge must also be deduced.

$$H\_UserCoLocation(U_1, U_2, Room\ 1, T_1)$$
$$H\_UserCoLocation(U_1, U_2, Room\ 1, T_2)$$
$$\neg H\_UserCoLocation(U_1, U_2, Room\ 1, T_3) \quad (5.4)$$
$$(5.5)$$

The above example demonstrates that as high-level state changes in response to primitive state changes, e.g., user movements, there are no language symbols generated that represent such changes. That makes an FSM-based implementation infeasible.

State semantics that belong to this category include ***not exists, nobody, nowhere, not seen, absent, empty, idle,*** etc. In general, all language elements that negate knowledge from a source of context, in the model belong here.

Uncertainty Semantics. The *Closed World Assumption* can be introduced to provide a partial solution to this problem, in absence of a better solution. The Closed World Assumption advocates that *if you cannot prove P or $\neg P$ from a knowledge base KB add $\neg P$ to the knowledge base KB,* or in other words, assume that P is false. If the Closed Word Assumption is used to assume that lack of a predicate P means that the predicate is false, then often an inconsistent view of the physical world is obtained. Consider the case where an Active BAT is occluded from the sensors, and therefore the system has temporarily no information for this user. The Closed World Assumption can be used in order to create fail-safe systems, such as a fire-protection system that assumes there is a fire when a "heart-beat" event that notifies the system of the opposite case is missing. However, in sensor-driven systems using the Closed World Assumption, the system may assume that the user is absent which is false, because the user is actually only temporarily hidden. This introduces a degree of *uncertainty*. Event-based techniques do not offer semantics that can differentiate, e.g., between a real absence and absence that is due to temporary occlusion, i.e., they do not inform on the degree of uncertainty.

5.6.2 Partial Knowledge

A dual problem to the above is that of *partial knowledge* that is caused by the fact that FSMs lack memory structures in which previous state for the world can be stored. The state of a physical entity is only known to the FSM, only when the respective symbol that corresponds to a sensor reading for that state of the entity occurs for the first time in a string and is read by the FSM. This means that until a symbol occurs, the FSM has no knowledge about the predicate it represents. This means that only partial knowledge is maintained in the system at that time; however, there are no semantics to express partial knowledge, as the FSM can be in an accepting or a non-accepting state (see Appendix A). This often occurs after a startup or a reboot of the FSM. FSM reboots can be assumed to occur after a node that has failed reconnects to the network, after its own failure, or some network failure on its connecting link. Partial knowledge is illustrated in Appendix A with an example.

5.6.3 General FOL Expressions with Free Variables

The FSMs' inability to deal effectively with negation has a direct effect on transparent reasoning with state. State differs from events in that state may hold at the current instant and may not hold at a next instant while an event happens and cannot be undone. Thus, when reasoning with state before each transition from an FSM state to another, a test needs to take place in order to ensure that the system state which is represented by the FSM state still holds and hasn't been invalidated with respect to a sensor update. For example, when evaluating whether two users are co-located, the individual movements of each user may lead to the invalidation of the co-location, e.g., the second user may enter the common area after the first one has already left, in which case, there is no co-location.

On the other hand, *transparency* (see Section 5.2) advocates that what would constitute an acceptable solution would be finite state machines that accept expressions with free variables that would be bound only when applied to each model separately. This section demonstrates that known methodologies for parametric FSM-based reasoning [5, 39] are not directly applicable here. In those methods, a parametric expression is modelled with an FSM with free variables and for each free variable in the initial parametric FSM, an identical, non-parametric FSM is spawned whenever a symbol that instantiates the free variable occurs at a given state. In the spawned FSM, all instances of the parameter that corresponds to the symbol which has been encountered are substituted with the actual symbol. However, in contrast to an event that occurs in a deterministic way, a state can be activated and de-activated, in response to any received event. Therefore, each state S of the parametric FSM that models P needs to have a transition to a state S' that models $\neg P$ and which cancels the execution. Unless $\neg P$ can be directly extracted from a primitive event, $\neg P$ does not exist as a symbol in the system. The *Closed World Assumption* can be used to determine a set of states $\{Q_i, i = 1, 2, \cdots\}$ where $\neg P$ can be assumed to hold. This is possible only if $\{Q_i, i = 1, 2, \cdots\}$ is a set of states that represent concrete or deduced knowledge. Even so, creating the transitions from S to $\{Q_i, i = 1, 2, \cdots\}$ requires knowledge of the underlying universe of discourse, which makes the reasoning *non transparent*. Figure 5.1 illustrates this with an example. In Figure 5.1(a), if an event occurs that corresponds to user a_1 being inside region r_1, then the automaton of Figure 5.1(b) will be spawned from the one in (a). In this automaton, the transitions from s_1 to s_3 must represent all events that report user a_1 exiting from region r_1. This means that there must exist one transition for each region in the universe which is different from r_1. If instead user a_1 had moved into r_2, the automaton that would need to be constructed, as a result, would be that of Figure 5.1(c).

Appendix A.4 contains a detailed illustration of this issue, using examples. Phrases that belong to this category include semantics with the meaning of somebody, somewhere, anybody, anywhere, everybody everywhere, exists that correspond to universal and existential quantification, with the exception of negative existential quantification ($\nexists$) that leads to lack of information and cannot be directly implemented with FSMs.

Dynamic Extensibility. The Closed World Assumption often leads to state explosion. For example, a co-location of 2 users in a room can be caused by either the sequence: *user A is in the room and then user B is in the room and user A hasn't left the room before user B has entered the room*, or the other way round (Figure 5.2). As the size of the physical world grows, the FSM grows exponentially. For example, assuming that user a_3 is added to the previous closed world, the FSM of Figure 5.2 would need to be re-compiled to that of Figure 5.3.

5.6.4 Size

Finite state machines are generally very large, as they need to take into consideration all the possible ways that a state can be reached. Figures 5.2 and 5.3 illustrate this.

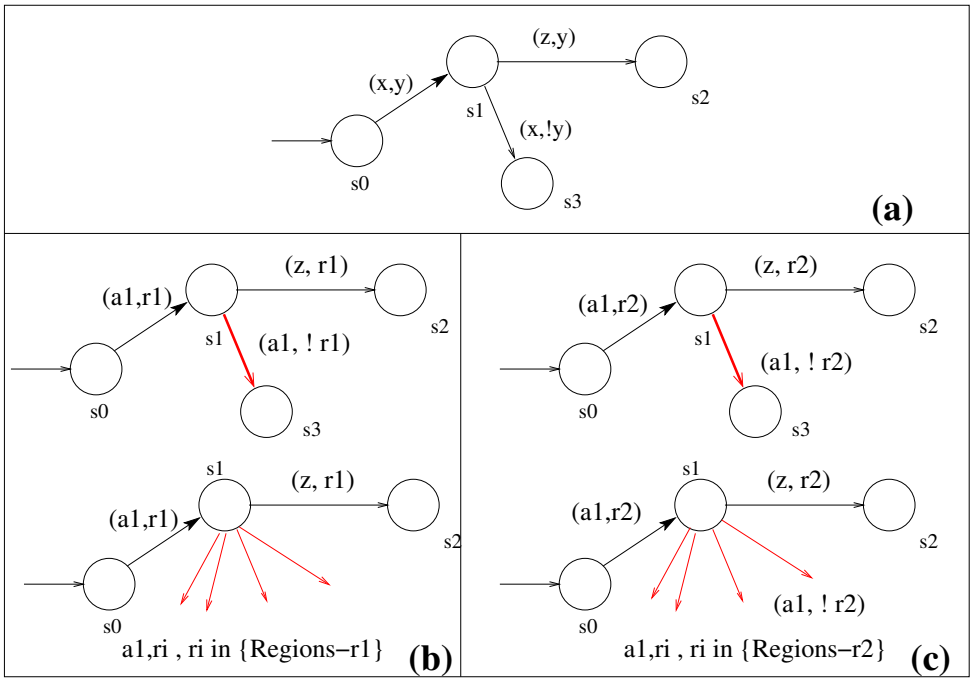


Figure 5.1. "Any two users are co-located".

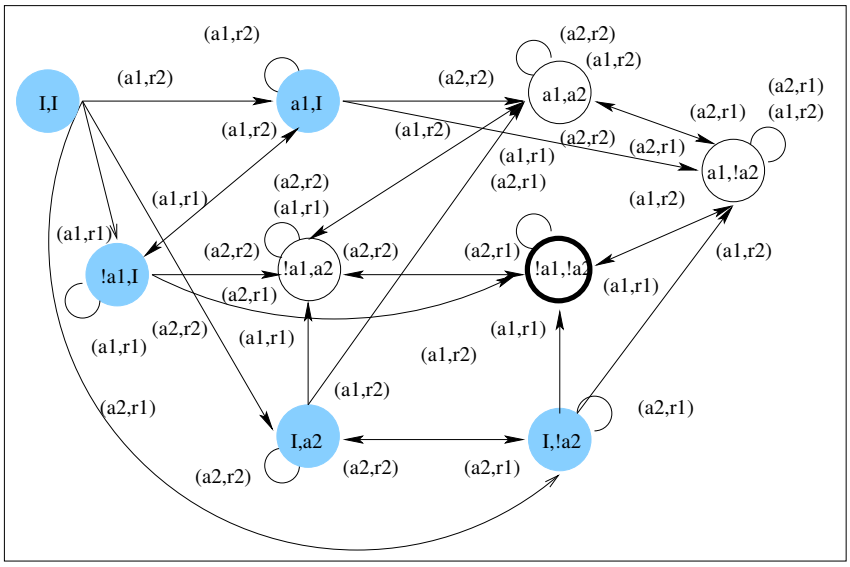


Figure 5.2. "Everybody is in r_2" with 2 users.

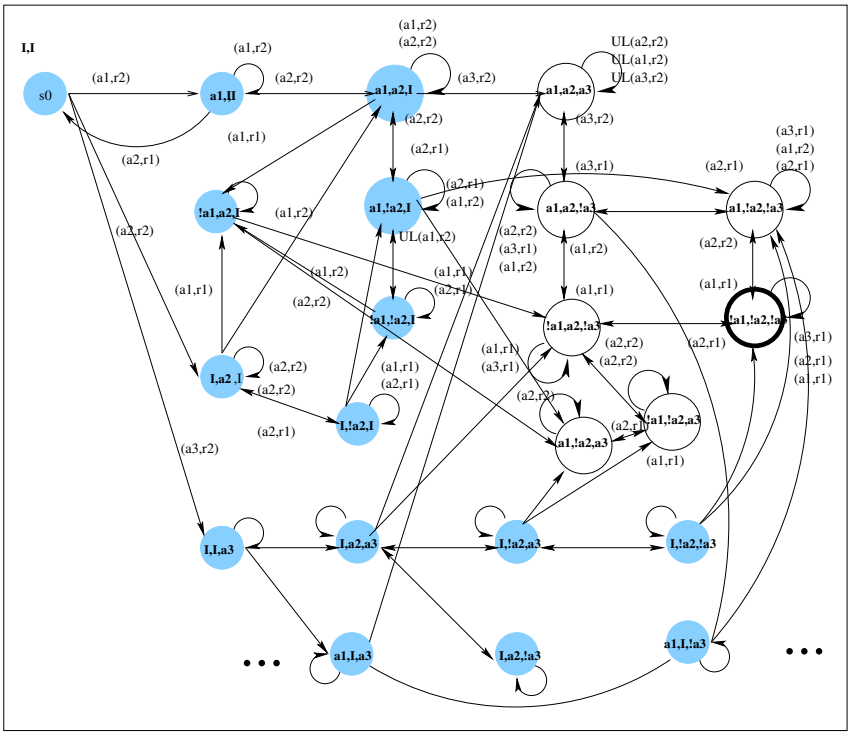


Figure 5.3. "Everybody is in r_2" with 3 users.

5.6.5 Conclusions

As can be concluded from the above analysis, event-based approaches are not sufficient for modelling context-awareness in sensor-driven systems in a way that the requirements of Section 5.1 are satisfied. The rest of this chapter presents a formally defined alternative model that satisfies the specified requirements. The proposed model is state-based and it integrates knowledge in the form of TFOL predicates. It is also compatible with the statistical model presented in Chapter 3. Temporal issues are discussed in Chapters 8 and 12.

5.7 A Model in First-Order Logic

In this section we present a formally defined model based on first-order logic for sensor-driven systems. We have adopted a state approach in modelling Sentient Computing systems. The model is compatible with the requirements of Section 5.1 by demonstrating the following properties:

- It stores aggregated knowledge about the state of the physical world in a deductive knowledge base. Knowledge is in the form of TFOL predicate instances (facts).
- It is compatible with the inferencing methodologies of Chapters 3 and 4. This is achieved by providing predicates for the movement phonemes and semantics to express and reason with likelihood. It also offers support for modelling confidence levels that evaluate the statistical inferencing.
- It is scalable. Scalability is achieved by distinguishing between concrete and abstract knowledge. It is dynamically extensible. Dynamic extensibility is achieved by supporting application-driven deduction of abstract state from concrete state using TFOL as a reasoning language. Chapter 6 discusses scalable, abstract reasoning.

- It supports asynchronous communication through the interface discussed in Chapter 8.
- It is generic enough to be independent of the sensor technology and the specific topology and consistency in terms of contained objects of the physical environment. It can be transported to different physical environments.

5.7.1 Model Life-Cycle

A sensor-driven system model [65] consists of three elements P, Q, D. P is the set of *primitive state predicates* that represent a sensor-driven domain. Q is the set of all instances of the primitive state predicate types of P. D is the *universe of discourse* of the domain.

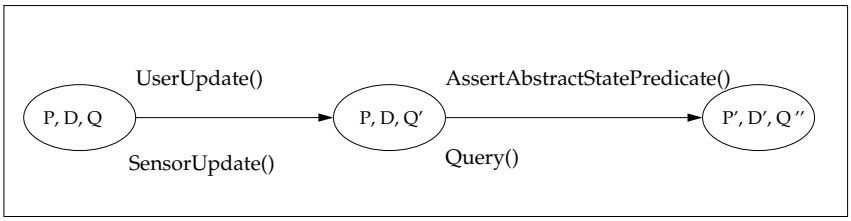


Figure 5.4. Sensor-driven system model cycle.

Figure 5.4 shows the stages the model goes through as it receives updates from the sensor infrastructure (SensorUpdate() interface) and the user (UserUpdate() interface). Such updates add new instances of the predicates in P to the model, thus modifying Q. Let Q' be the current new, modified set of instances of P, after an update. The interface AssertAbstractStatePredicate()[1] causes the creation of an *abstract state predicate* in P. P' is the extended set of predicates that results.

5.8 Model Definitions

Temporal first-order logic (TFOL) was chosen as being appropriate and sufficient for the description of the model. We assume that the physical world contains N individual values that represent autonomously mobile objects such as people that work in a building. We also assume that the physical environment contains K individual values which represent known physical *locations* of interest. Locations can be classified into *atomic* locations and *nested* locations. *Atomic* locations will typically be polyhedral named regions such as *"the coffee-area"*, *"mike's desk"*and rooms. Via a process of *nesting* we produce a set of aggregated polyhedral regions such as floors and buildings (each floor may contain a specific set of rooms and each building a particular set of floors) as well as logically aggregated spaces such as departments (each department may contain a number of floors, or buildings).

We define a set P of first-order logic predicate sets that represent a generic sensor-driven spatial domain.

$$\begin{aligned} P \;=\; & \{SelfMobileLocatable, NonSelfMobileLocatable, AtomicLocation, \\ & NestedLocation, InRegion, UserMovement, \\ & \{Functions\}, \{Higher\text{-}Order\ Functions\}\}. \end{aligned} \tag{5.6}$$

[1]The interface AssertAbstractStatePredicate(AESL definition, filter, type definition) corresponds to the interface Subscribe() of Figure 1.1. It is invoked automatically by the SCAFOS framework, in response to a user creating a new context-aware application using SCALA, and it contains the required high-level specifications (AESL definition, filter, type definition) that will be used by SCAFOS in order to generate the abstract predicate that is of interest to the newly created application. This is discussed in detail in Chapters 8 and 12.

The above predicates are explained in detail in Section 5.8.1. Furthermore, we define a TFOL *model* [65] M of P that consists of a *logical domain* D of the following subsets, as well as a *mapping* of P to D. Furthermore, we assume that in a distributed architecture that consists of more than a single sensor-driven component, there is one logical domain for each spatial sensor-driven component.

$$D = \{Sensors, SelfMobileLocatables, NonSelfMobileLocatables, Positions, Regions, Roles, RegionalAttributes\}.$$

The subset *Sensors* contains all the known sensor entities in the spatial domain of interest. Similarly, the subset *SelfMobileLocatables* contains all humans in this domain, and the subset *NonSelfMobileLocatables* contains all inanimate objects in the domain. The subset *Regions* contains all known regions in the domain such as rooms, regions within a room and larger regions such as floors, labs and buildings. The subset *Roles* contains all known roles that a user can be associated with in that domain, e.g., *System Administrator, Supervisor, Phd (student)*, etc. The subset *RegionalAttributes* contains a number of *keywords* that are used to characterise several properties of regions, such as their functionality/ownership, etc. This allows for the specification of *semantic queries*. For example, semantic queries can be specified that locate all known meeting rooms or the closest system administrator. Queries are discussed in Chapter 6. The mapping between P and D is achieved through the predicates of P.

5.8.1 Locatables

$SelfMobileLocatable$ is a predicate on the set $\{SelfMobileLocatables \times Roles\}$ and it represent objects that can be located and that can move on their own, such as humans. Its format is depicted below:

$$SelfMobileLocatable(id, role),$$
$$id \in SelfMobileLocatables, role \in Roles.$$

$NonSelfMobileLocatable$ is a predicate on the set $\{ NonSelfMobileLocatables \times Roles\}$, which represents objects. Its format is described below:

$$NonSelfMobileLocatable(id, role),$$
$$id \in NonSelfMobileLocatables, role \in Roles.$$

5.8.2 Spatial Abstractions

Spatial abstractions are represented by means of the predicates $AtomicLocation$ and $NestedLocation$.

An $AtomicLocation$ predicate is a ternary predicate on the set $RegionalAttributes \times Positions^m\}$, with the following variables:

$$AtomicLocation(rid, rattr, polygon),$$
$$rid \in Strings, rattr \in RegionalAttributes,$$
$$polygon = \{\langle x_1, y_1, z_1\rangle \cdots, \langle x_m, y_m, z_m\rangle\}, \langle x_i, y_i, z_i\rangle \in Positions.$$

A $NestedLocation$ predicate is a ternary predicate on the set $\{RegionalAttributes \times Locations^m\}$ with the following format:

$$NestedLocation(rid, rattr, list\text{-}of\text{-}contained\text{-}locations),$$
$$rid \in Strings, rattr \in RegionalAttributes,$$
$$list\text{-}of\text{-}contained\text{-}locations = \langle l_1, \cdots, l_k\rangle, l_i \in Regions.$$

Function name	**Description**
Probability	$Prob(value, confidence\_level, predicate, time\_reference)$
Speed	$Speed(value, uid, time\_reference\ )$
Distance	$Distance(value, uid, role, rid, rattr\ )$
User-Distance	$UDistance(value, uid_1, role_1, uid_2, role_2)$
MostProbable	$MostProbable(value, confidence\_level, class\ variable, predicate, time\_reference)$

Table 5.1. Functions

5.8.3 Functions

A set of first-order logic function predicates are predefined in the model. A function predicate P_f is an internal representation of a function f from a set $\{A \times B\}$ to a set C with a predicate P_f on the set $\{A \times B \times C\}$, like Prolog. As a convention, the first argument ($value$) represents the result of the function.

Function predicates take as arguments both primitive and abstract predicates. The most representative are seen in Table 5.1.

The *Probability* function predicate represents likelihood estimations, such as the ones that are generated by Bayesian prediction (see Chapter 4). As the analysis can be applied to any predicate in the model, this function takes as arguments the predicate of interest ($predicate$). The variable $value$ holds the value of the estimated likelihood for this predicate instance and the variable $confidence\_level$ holds the value of the confidence level for that estimation (see Chapter 4). The variable *time_reference* is of type string and is used to hold the value of the temporal reference against which the probability is estimated. For example, if it is estimated that the likelihood that the assumption that the supervisor will be at his office today is 85%, and the confidence level for this probability estimation is 80%, this can be expressed as:

$$Prob(85\%, 80\%, H\_UserInLocation(uid, \textit{Supervisor}, rid, \textit{Supervisor's-office}), Today).$$

The $MostProbable$ function predicate stores the value of the classification score for the prediction class, the predicate on which the Bayesian network that performs the prediction is based as well as a set of predictor variables (see Chapter 4). For example, "the most probable location where a user will be seen today" is expressed as follows:

$$MostProbable(value, confidence\_level, H\_UserInLocation(uid, rid, role, rattr, Today).$$

The $Distance$ function is a function on the set $SelfMobileLocatables \times Positions \times Regions$. It contains a value parameter which stores the value of the distance of a user's position from the centre of a given region. The $UserDistance$ function behaves similarly to $Distance$ but calculates the distance between two users instead of a user and the centre of a region. The mathematical representation for this function predicate[2] given two points $A(x_1, y_1, z_1), B(x_2, y_2, z_2)$, where point A represents a user position and B represents a reference point (the centre of a region), is:

$$Distance(a, b) = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}.$$

The $Speed$ function predicate is a function predicate on the set $SelfMobileLocatables$ and the set of $Positions$. The user's speed is calculated by two successive sightings and is held in the variable $value$. The mathematical representation of this function predicate[3] given two successive user sightings

[2]In practice, most context-aware applications only need a 2-D distance representation and for this reason the z coordinate has been omitted from this implementation.

[3]In practice, most context-aware applications only need a 2-D speed representation and for this reason the z coordinate has

$A(x_1, y_1, z_1, t_1), B(x_2, y_2, z_2, t_2)$ where $t_2 > t_1$ and $\delta t = t_2 - t_1$, is:

$$Speed(t_2) = \frac{\sqrt[2]{(x_2 - x_1)^2 + (y_2 - y_1)^2}}{\delta t}.$$

5.9 Conclusions

A state-based model in TFOL was presented in this chapter. State-based modelling was demonstrated to be both essential and beneficial for context-awareness in real-time, distributed, sensor-driven systems. The model is well-defined, uncertainty is included in the predicate semantics and it offers powerful tools for mathematical and statistical reasoning.

been omitted from this implementation.

Chapter 6

Knowledge-Representation and Scalable Abstract Reasoning for Sentient Computing using First-Order Logic

The previous chapters introduced a state-based conceptual framework for aggregating knowledge in Sentient Computing. This chapter presents a dynamic knowledge base maintenance system for representing and reasoning with knowledge about the Sentient Computing environment based on the model of Chapter 4. Sentient Computing has the property that it constantly monitors a rapidly changing environment, thus introducing the need for abstract modelling of the physical world that is at the same time computationally efficient. The approach in this chapter uses *deductive systems* in a relatively unusual way, namely, in order to allow applications to *register inference rules* that generate *abstract* knowledge from low-level, sensor-derived knowledge. *Scalability* is achieved by maintaining a *dual-layer* knowledge representation mechanism for reasoning about the Sentient Environment that functions in a similar way to a two-level cache. The lower layer maintains knowledge about the current state of the Sentient Environment at sensor level by continually processing a high rate of events produced by environmental sensors, e.g., it knows of the position of a user in space in terms of his coordinates *x,y,z*. The higher layer maintains easily retrievable, user-defined, abstract knowledge about current and historical states of the Sentient Environment along with temporal properties such as the time of occurrence and their duration. For example, it knows of the room a user is in and how long he has been there. Such abstract knowledge has the property that it is updated much less frequently than knowledge in the lower layer, namely, only when certain threshold events happen. Knowledge is retrieved mainly by accessing the higher layer, which entails a significantly lower computational cost than accessing the lower layer, thus maintaining the overall system scalability. This is demonstrated through a prototype implementation.

6.1 Scalable Abstract Reasoning

Chapter 5 proposes a model for sensor-driven systems that supports both concrete and abstract knowledge. Abstract knowledge is used by Sentient Computing applications, that can be viewed as a logical layer, namely, the *Application Layer* in the *Sentient Applications layered architecture* (Figure 6.1). Concrete knowledge is produced by the *Sensor Infrastructure Layer*. As discussed in Chapter 5, there is a significant *gap* between the level of abstraction in the *knowledge* about the Sentient World that Sentient Computing applications require for their functionality and the actual *low-level data* that are produced by the sensors and which constitute a *low-level, precise, knowledge layer.* For example, an application that displays the user's screen in response to his proximity to his PC needs to know when a more abstract situation has occurred, that is, when the user is close to his PC. The information about the user's proximity

to his PC is a logical abstraction of his position in space, and it is expressed in relation to the position of another physical object, namely, his PC. To make matters worse, the above system will need to monitor a large number of users distributed among a number of distinct locations at the same time. Even so, it needs to react to the perceived changes with no perceptible delay.

This chapter proposes that the gap between the application-layer abstraction and the sensor-derived precision be bridged by using a *deductive* component that reasons with low-level, sensor-derived knowledge in order to *deduce* high-level, abstract knowledge that can, in turn, be used easily by the application layer. Furthermore, the proposed *deductive reasoning* does not compromise computational efficiency and performance. For very large distributed environments, there is scope for further research.

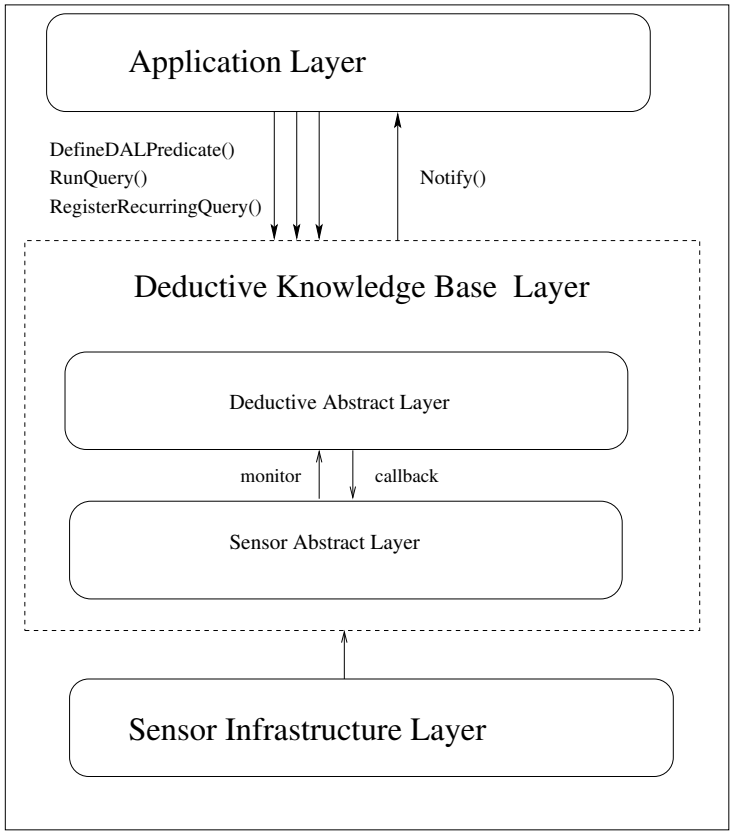


Figure 6.1. The Sentient applications layered architecture and its API.

This chapter tackles the above issue of *scalable, system-level, computationally efficient abstract modelling* of the physical world. Its contributions are a formal definition of a knowledge representation as well as a mechanism for reasoning with such knowledge using *logical deduction* that combines *expressiveness*, *scalability* and *performance*. The proposed approach generalises previous efforts to abstract knowledge from sensor data that have resulted in limited, case-specific abstractions, such as the ones supported by SPIRIT. SPIRIT's world model contains the notion of abstracting regions from Active BAT positions. QosDREAM supports abstractions for entering and leaving a region.

6.1.1 Layered Interfaces

For the abstract model of the Sentient world, a dual-layer knowledge representation architecture was designed. This design approach is inspired by the *OSI paradigm* [77] for layered network architecture,

where each layer incorporates a set of similar functions and hides lower-level details from the layer above it, thus achieving simplicity, abstraction and ease of implementation.

6.2 Knowledge Representation

This section describes the architecture of the two logically distinct layers of knowledge representation and discusses their functionality as a two-layered cache for the Sentient Application Layer.

For the needs of this section, a definition introduced by Samani and Sloman in [63] is used, according to which an *event* is a happening of interest that occurs instantaneously at a specific time[1]. Furthermore, the *Sentient environment* is defined as the *physical* environment, and the *current logical state* of the Sentient environment is defined to be the set of all known facts about the Sentient environment between an *initial event* and a *terminal event*. *Initial* and *terminal* events can be any events that are of interest to the Sentient application layer. Based on the above, we can say that the *Sensor Abstract Layer* maintains a *low-level* but *precise* view of the *current logical state* of the Sentient environment, as produced by sensors that are distributed throughout the environment and continually updated through events. Equally, we can say that the Deductive Abstract Layer maintains an *abstract* view of the *current logical state* of the Sentient environment.

Particularly interesting sources of events are those that characterise the *location* of an object. These are generated by a *location system* such as the Active BAT [41], where the positions of users in 3-D space are tracked, typically once per second, by means of an ultrasonic transmitter called BAT. The *Sensor Abstract Layer* processes all the generated events, and thus knows of the *last position* of all users in the system in terms of their coordinates.

A more abstract view about the state of the Sentient environment can easily be *inferred* from the knowledge stored in the Sensor Abstract Layer. For example, from a user's position (x, y, z) and from a set of known polyhedra that represent regions, the room the user is in follows logically. Furthermore, from a known set of nested polyhedra, additional locations in which the user is present can also be inferred[2].

The *Deductive Abstract Layer* (DAL), through its interaction with the Sensor Abstract Layer (SAL), maintains such abstract knowledge about the *current* and *past* states of the Sentient Environment together with temporal information about the *initial events* that triggered them, and the duration of each state. Such data can be used by statistical models in order to generate a likelihood estimation of situations that may occur in the future, based on their past occurrences [52].

The two layers interact through a monitor-callback communication scheme with the help of a *TFOL formula* that is used as a specification (see Chapter 12, AESL definition). A *monitor call*, initiated by the Application Layer, causes the Sensor Abstract layer to *filter through* to DAL only those low-level changes that affect the abstract knowledge stored in the Deductive Abstract Layer, thus relieving the Deductive Abstract Layer from the cost of continually monitoring all the data that are produced by the sensors. Consequently, knowledge in DAL is updated at a significantly lower rate than it does in SAL, ensuring that large amounts of physical data can be processed by replicated SALs, maintaining at the same time the overall system scalability.

6.2.1 The SAL-DAL API

The application Layer communicates with the dual Deductive Knowledge Base Layer via an API consisting of ***DefineDALPredicate()***, ***RunQuery()*** and ***RegisterRecurringQuery()*** interfaces (see Figure 6.1).

[1] Chapter 8 introduces a general concept of an event, an *abstract event*.

[2] The query: "Is user X in Cambridge?" needs to answer positively even if User X is in FC15, which is in the William Gates Building in Cambridge.'

The *DefineDALPredicate()* interface takes as arguments the predicate name along with its parameters, and creates the necessary representation in the Distributed Abstract Layer (DAL) for this piece of knowledge. This is equivalent to a type definition in Chapter 12. The *RegisterRecurringQuery()*, *RunQuery()* and *Notify()* interfaces are discussed next.

Application-Driven Deduction

The *RegisterRecurringQuery()* command is used by the Application Layer in order to register interest in a particular, recurring situation in a way that the application layer is *notified* whenever the situation occurs, starting with its next occurrence. The *RegisterRecurringQuery()* command, together with the *Notify()* command, behave similarly to a *publish-subscribe* protocol. Chapter 8 discusses an extended publish/subscribe protocol for sensor-driven systems.

The interface between the two layers, used by both API commands, is based on a *monitor-callback* mechanism similar to an *asynchronous invocation* between a *consumer* and a *publisher*. Nomenclature here is taken from the theory of Distributed Systems [22]. The *monitor* mechanism in the Sensor Abstract Layer watches all changes in the environment reported by the Sensor Infrastructure for certain threshold events, as specified in the *RunQuery()* and *RegisterRecurringQuery()* statements. The *callback* mechanism ensures that such threshold events, when they occur, trigger a corresponding update in the Deductive Abstract Layer, creating instances of the predicates that hold and destroying any that are no longer true.

Example. In order to illustrate the functionality of the dual-layer architecture in more detail, consider the case where the Application layer is interested in receiving a notification whenever two or more users are co-located. Through a *RunQuery()* or *RegisterRecurringQuery()* statement initiated by the Sentient Application Layer, unless it already knows about co-located users, DAL will register a *monitor()* call to SAL in order for the latter to start monitoring the sensor data that signify co-location occurrences, as specified in the recurring query. As a result, the Sensor Abstract Layer monitors the incoming events in order to determine (from the users' positions) whether two or more users are contained in the same room. When this occurs, the respective knowledge about the users co-location will be generated in the Deductive Abstract Layer through a *callback()* call. All further changes in the position of these users in the Sensor Abstract Layer are monitored in order to determine whether the two users remain co-located. If any of the co-located users exit the containing region, the change in the users' location in combination with the co-location predicate instance in the current, abstract state (DAL), signals an inconsistency. As a result, another *callback()* call is triggered from SAL to DAL, invalidating the current state, logging it as a historical state and generating a new current state. In practice, only a fragment of the global state of the Sentient Environment is changed, as most abstract knowledge remains unaltered.

6.2.2 Scalability Concerns

The main benefit of the proposed architecture is that it maintains a consistent, abstract state of the Sentient environment in the Deductive Abstract Layer that can be made available to the application layer at a significantly lower cost than if it were generated directly from the Sensor Abstract Layer. The availability of the abstract knowledge in DAL, and the fact that this knowledge changes at a lower rate than it does in SAL, make DAL more computationally efficient at keeping its stored knowledge consistent. Figure 6.2 depicts the different rates with which knowledge is updated in each layer for two predicates P_1 and P_2. Section 6.5 discusses, in more detail, computational concerns associated with the functionality of the two layers. Note that λ^L is considerably greater than the rates λ^{H_1} and λ^{H_2}.

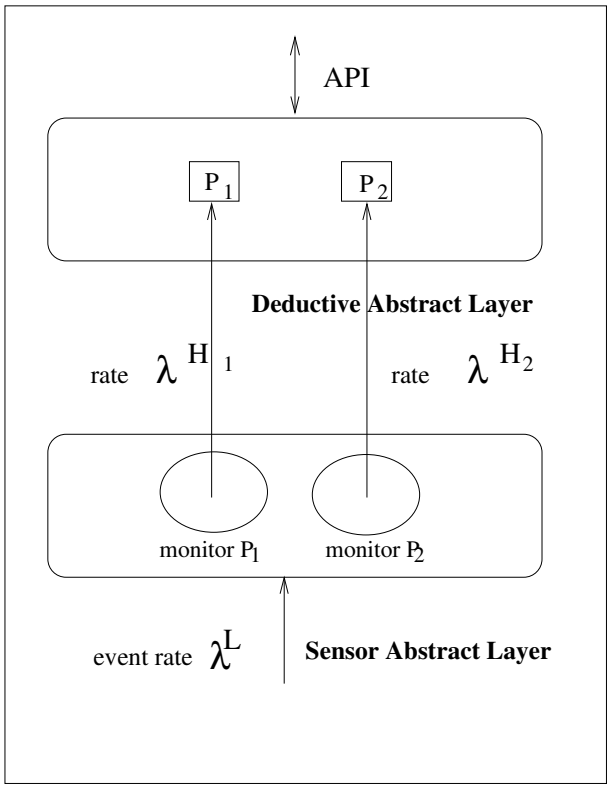


Figure 6.2. SAL-DAL

6.3 Formal Definition

This section presents a formal definition of the proposed scalable knowledge representation architecture for Sentient Computing. The concepts of the dual-layer architecture that were discussed in the previous section are now formally defined using first-order logic.

6.3.1 First-Order Logic

First-order logic [65], or predicate calculus, was chosen as being appropriate and sufficient for the description of the two knowledge layers, as they both maintain either current knowledge only (Sensor Abstract Layer) or a combination of current and historical knowledge (Deductive Abstract Layer) about the Sentient Environment. Time is implicit in SAL and explicit in DAL. However, when describing the monitoring mechanism that establishes the links between the two layers, temporal aspects of the described predicates are addressed by realising that as the Sensor Abstract Layer is updated first, until the changes are updated to the Deductive Abstract Layer, this will contain the last known abstract state of the world.

Concepts and Definitions A *knowledge base* KB is a system that stores knowledge about the Sentient environment. A knowledge base represents predicates that are *true* by storing an instance of each of these predicates. We refer to each such instance as a *fact*. The *assertion* of a *fact* in the knowledge base is equivalent to it being *stored* in the knowledge base as a *true* statement. A fact being *retracted* from the knowledge base results in the *removal* of the fact from the knowledge base. In fact, the *assert* command is similar to a database ADD whereas the *retract* command is equivalent to a database DELETE. When a fact is *asserted* in the knowledge base, this signifies that the predicate that the fact corresponds to has the value TRUE. When the fact is *retracted* from the knowledge base, this signifies that the corresponding predicate has the value FALSE. This nomenclature is taken from logic programming.

6.3.2 Naming Convention for Predicates

For reasons of clarity and simplicity, the following naming convention was adopted for logical predicates throughout this document:

L_⟨*SAL predicate name*⟩ ((argument name ?*argument value*)$\cdots$ (argument name ?*argument value*))
H_⟨*DAL predicate name*⟩ ((argument name ?*argument value*)$\cdots$ (argument name ?*argument value*))

The main difference is that DAL predicates have additional time parameters that represent the beginning and, wherever appropriate, the end of the temporal situations to which they refer. For the description of the predicates, a *named parameter notation* was used based on the CLIPS [19, 20] syntax. Table 6.1 portrays some significant predicates. Each *predicate argument* has an associated *value*, which is denoted by *?argument-value*. The predicates and their arguments are discussed in detail in sections 6.3.3 and 6.3.4.

6.3.3 Sensor Abstract Layer (SAL)

The knowledge base of this layer contains up to N facts of type *L_UserAtPosition(uid,role,x,y,z)* [3] that represent an object's last known position in 3-D space in terms of its Cartesian coordinates x, y, z. $L\_UserAtPosition$ is the most precise location known to the system for each user. The variable *?uid*

[3] The positional parameters notation *L_UserAtPosition(uid,role,x,y,z)* is used interchangeably with the named parameter notation *L_UserAtPosition*(uid *?uid*)(role *?role*)(x *?x*)(y *?y*) (z *?z*)). The former is used mainly for clarity in predicate description; the latter is used mainly to describe the implementation.

	Current Predicates	Historical Predicates
DAL	**(H_UserInLocation**(uid ?uid)(role ?$role$) (rid ?rid)(rattr ?$rattr$) (start-time ?*time-value*)) **(H_UserColocation**(uid-list ?$uid_1 \cdots ?uid_n$) (rid ?rid)(role-list ?$role_1 \cdots ?role_n$) (rattr ?$rattr$)(start-time ?*time-value*)) **(H_UserIsPresent**(uid ?uid)(role ?$role$) (start-time *?time-value*))	**(H_ UserInLocationHistoric**(uid ?uid) (role ?$role$) (x ?x)(y ?y) (z ?z)(rattr ?$rattr$) (start-time *?time-value*)(end-time ?*time-value*)) **(H_UserColocationHistoric**(uid-list ?$uid_1 \cdots ?uid_n$) (role-list ?$role_1 \cdots ?role_n$) (rid ?rid) (rattr ?$rattr$) (start-time *?time-value*) (end-time ?*time-value*)) **(H_UserIsPresentHistoric**(uid ?uid)(role ?$role$) (start-time *?time-value*)(end-time ?*time-value*))
SAL	**(L_UserAtPosition**(uid ?uid)(role ?$role$) (x ?x)(y ?y)(z ?z))	-

Table 6.1. Naming convention for logical predicates.

represents the unique user identification for that particular user. In examples in this chapter, first names of users are used as identifiers. The variable *?role* represents the user's role, e.g., Supervisor. The variables *x,y,z* represent the user's last known coordinates. In this way, each user is associated with a position in space.

Apart from these positions, the knowledge base also contains M_1 facts of type *L_AtomicLocation* to the M_1 known atomic regions of physical space, (e.g., rooms and polygonal areas of space). The KB also contains M_2 facts of type *L_NestedLocation*, each corresponding to M_2 nested regions (floors, larger areas, buildings, neighbourhoods). There are, therefore, four distinct type of predicates represented in this layer.

(*L_UserAtPosition* (uid *?uid*)(role *?role*)(x *?x*)(y *?y*) (z *?z*))
(*L_AtomicLocation* (rid *?rid*)(rattr *?rattr*) (polygon ?n_1?$n_2 \cdots ?n_j$))
(*L_NestedLocation* (rid *?rid)* (site-list ?$site_1 \cdots ?site_k$))
(*L_InRegion* (x *?x*)(y *?y*)(z *?z*)(rid *?rid*)(rattr *?rattr*))

As the people move in space, a location system generates, on average, λ^L *L_UserAtPosition* facts/sec per mobile user and asserts them in the knowledge base. For each new fact of type *L_UserAtPosition*, the fact that represented the previous known position for that user is *retracted*, so that the knowledge base only contains the most recent known location for that user.

The predicate *L_AtomicLocation* associates a named location such as "Room 5" characterised by a unique identifier (the *rid*), with a set of j points, $n_1 \cdots n_j$, which form the nodes of a polyhedral region that defines that area.[4] The predicate *L_NestedLocation* associates a *nested location* such as "The Computer Laboratory" with a list of nested and atomic locations that are directly contained in it. The predicate *L_InRegion* is created as a result of a spatial indexing algorithm that determines the *smallest* region that contains the given coordinates, as expressed in the *L_UserAtPosition* predicate.

6.3.4 Deductive Abstract Layer (DAL)

The higher level is logically distinct from the lower level in that it maintains a complete view of the Sentient world. Although it lacks knowledge of the accuracy of the exact user position (as this is only known to the Sensor Abstract Layer), it knows of high-level situations seen from a user-perspective as well as their temporal properties, i.e., whether they hold at the current instant, or whether they happened in the past, when they first occurred and what was their duration. Such *dynamic knowledge* is modelled in the form of *current* and *historic predicates*. *Current* predicates represent a *dynamic situation* that still

[4] A coordinate system is assumed that assigns a set of coordinate values *x,y,z* to each position in space.

holds. *Historic* predicates represent a *situation* that occurred for a certain interval, beginning at a certain point in time and ending at a later point in time. As a consequence of the above modelling technique, the Deductive Abstract Layer has the important property that it gradually accumulates information about what has happened in the Sentient world. The format of the DAL predicates is discussed in detail below.

The DAL Current Predicates. *DAL current* predicates describe a situation that occurred at an instant t_0 and that still holds at the current instant, (which is represented by the value *now*). Such predicates have the following general format:

(*predicate name* (arg_1 $?arg_1$)$\cdots$ (arg_n $?arg_n$) (start-time *?time-value*))

Arguments arg_1 to arg_n represent the parameters of the situation that is described by the predicate, and the variables $?arg_1$ to $?arg_n$ their respective values. The argument named "start-time" represents the time when the situation described by the above predicate became first known to the system.

An important *current* predicate is the one used to describe a high-level location, e.g., Mary being in the proximity of the coffee-machine, or James being on floor 4.

(*H_UserInLocation*(uid Mary) (role PhD)(rid Coffee Area) (rattr Coffee Area) (start-time 11:02))

(*H_UserInLocation*(uid James) (role PhD)(rid Floor 4)(rattr Upper Floor) (start-time 13:05))

where *?uid* represents the user's unique identification and *?rid* is the value of the named parameter rid, which represents the name of the smallest region that contains the user.

Similarly, applications can request through the API for SAL to register their interest in situations where two or more users are co-located in the same high-level region by using the predicate *H_UserCoLocation*.

(*H_UserCoLocation*(uid-list $?uid_1 \cdots ?uid_n$)(role-list $?role_1 \cdots ?role_n$)(rid $?rid$)(rattr $?rattr$)
(start-time *?time-value*))

This process is explained in more detail in section 6.5. In the above formula, *uid-list* is the list of users that are co-located in a region with name rid. The variables $?uid_1$ to $?uid_n$ represent the unique identification of these users.

The DAL Historical Predicates. The DAL historical predicates describe a situation that occurred at a time instant t_0[5] , was true for a duration d and ceased to be true at a time instant t_1. Such predicates are expressed in the following general format:

(*predicate name* (arg_1 $?arg_1$)$\cdots$ (arg_n $?arg_n$) (start-time *?time-value*) (end-time *?time-value*))

The argument "start-time" represents the time when the situation described by the above predicate became first known to the system. The argument "end-time" represents the time when the situation stopped being true, e.g., when the user left the room. For example, the DAL historical predicate that describes the situation where Jane and Mike move into the meeting room in their office building at 12:46 pm, remain in the same room for 9 minutes and Jane leaves the meeting room at 12:55, is expressed

[5]The time is set according to the local clock of the Deductive KB component. Temporal issues are discussed, in detail, in Section 12.3.1.

below: It is worth noting that there can be multiple instances of historic predicates for the same user.

(*H_UserLocationHistoric* (uid-list Jane)(role-list PhD)(rid Room 9)(rattr Meeting Room) (6.1)
(start-time 12:46) (end-time 12:55))
(*H_UserCoLocationHistoric* (uid-list Mike Jane)(role-list PhD)(rid Room 9)(rattr Meeting Room)
(start-time 12:46) (end-time 12:55))

6.3.5 User-Defined DAL Predicates

It is worth noting that whereas all SAL predicates are predefined, all predicates in DAL are *user-defined*. This means that in the initial state, DAL contains no predicates. The structure of DAL predicates is defined through the *DefineDALPredicate()* API call (see section 6.2.1), which is similar to a *type definition* in the nomenclature of programming languages. *Instances* of these predicates (facts) are generated from the Sensor Abstract Layer by the *monitor()* and *callback()* calls (see Section 6.2.1).

6.4 Queries

Queries are used by the application layer in order to capture and return the current instance of stored knowledge about the Sentient World. *Queries* are similar to SQL [27] SELECT statements in the theory of relational databases. A ***query*** can be viewed as a first-order logical expression $f(c_1, c_2, \cdots, c_n)$ that has the ***property*** that upon the satisfaction of a set of *atomic formulae* $c_1, c_2, \cdots, c_n$, an *answer* is triggered.

$$f(c_1, c_2, \cdots, c_n) \Rightarrow Answer$$

where f is any first-order formula involving the formulae $(c_1, c_2, \cdots, c_n)$.

Answer can have a value of "yes", "no", "I don't know" or a value extracted from a stored fact such as the user id. The | operator is used in order to define the arguments whose values are to be returned and it is conceptually similar to an SQL *SELECT* operator. The interface through which the *answer is returned* to the user is subject to the application layer and can be implemented in various ways, e.g., by using a *print* function, by publishing a *structured event* or through an API. A *structured event approach*, where the answer is encoded as a *structured event* and is returned to the application layer via a *Notify()* call (see Section 6.2.1), is adopted. Event-based asynchronous communication of changes in knowledge predicates is discussed in Chapter 8.

Examples of logical queries are *"Who is present in the building now?"* and *"Which users are co-located now?"* The first query may be useful in the case of an application that delivers reminders to anybody who is present in the building late in the evening, in order to remind them to lock their door on the way out. The second query may be useful for the same application, delivering a reminder to one party which is semantically associated with the second, e.g., the reminder: "Remember to ask Jane to return your book" will be delivered when the user is in the same room with Jane [30]. Equally interesting as an example is the case where a user enters a conference site and is interested to know if there is anyone present from the University of Cambridge. If the above mentioned query *"Who is present in the building?"* was to be executed at a knowledge base with a single layer of knowledge, (i.e., the Sensor Abstract Layer), it could then be written as a query of the following form:

Query 1. *Return All Present Users (Sensor Abstract Layer).*

$uid|$(*L_UserAtPosition*(uid *?uid*)(role *?role*) (x *?x*) (y *?y*) (z *?z*))$\wedge$
(*L_AtomicLocation*(rid *?rid*)(rattr *?rattr*)(polygon $?n_1?n_2 \cdots ?n_j$))$\wedge$
(*L_InRegion*(x ?x) (y ?y)(z ?z) (rid *?rid*)(rattr *?rattr*))

In this case, *uid* represents the information that will be returned in the answer. Query 1 expresses the logical statement that in order for a user to be present in the building, three *conditions* need to hold simultaneously:

- He or she needs to be *seen* by the location system at a position that can be characterised by the coordinates x, y, z.
- The system must know of at least one region with id rid that can be characterised by a known polyhedral shape.
- The system is able to determine that the coordinates of the user's position belong to a known region such as the one described above.

If all of the above hold simultaneously, than the user is deduced to be *present*.

The same query, should it be applied on DAL, would assume a simpler form:

Query 2. *Return All Present Users (Deductive Abstract Layer).*

$uid|$(*H_UserIsPresent*(uid $?uid$) (role $?role$)(start-time *?time-value*))

Similarly, the query *"Which users are co-located now?"* can be viewed as:

Query 3. *Return All Co-Located Users (Sensor Abstract Layer).*

$(uid_1, uid_2)|$ (*L_UserAtPosition*(uid $?uid_1$)(role $?role_1$)(x $?x_1$)(y $?y_1$) (z $?z_1$))$\wedge$
(*L_UserAtPosition*(uid $?uid_2$)(role $?role_2$)(x $?x_2$)(y $?y_2$) (z $?z_2$))$\wedge$
(*L_AtomicLocation*(rid ?rid)(rattr $?rattr$) (polygon $?n_1?n_2 \cdots ?n_j$))$\wedge$
(*L_InRegion*(x $?x_1$)(y $?y_1$)(z $?z_1$) (rid $?rid$)(rattr $?rattr$))$\wedge$
(*L_InRegion*(x $?x_2$)(y $?y_2$)(z $?z_2$)(rid $?rid$)(rattr $?rattr$))$\wedge$
($uid_1 \neq uid_2$)

The same query, should it be applied on the Deductive Abstract Layer instead of the Sensor Abstract Layer, assumes a simpler form.

Query 4. *Return All Co-Located Users (Deductive Abstract Layer).*

$(uid_1, uid_2)|$(*H_UserCoLocation*(uid-list $?uid_1?uid_2$)(role-list $?role_1?role_2$)
(rid $?rid$)(rattr $?rattr$)(start-time *?time-value*))

Queries 1 and 3 are defined by the application layer to be equivalent to Queries 2 and 4, respectively (see AESL definitions, Chapter 12). However, Queries 2 and 4 have, on average, fewer conditions than their equivalent Queries 1 and 3. This is due to the fact that the information of the users' *presence* and *co-location* is available in the Deductive Abstract layer in the form of the logical predicates *H_UserIsPresent* and *H_UserCoLocation*, respectively. Section 6.6 discusses in detail the effect of the above observation on the computational complexity involved in the execution of queries in the proposed reasoning system, demonstrating that queries executed in the Deductive Abstract Layer, such as Queries 2 and 4, entail the use of significantly fewer computational resources than queries executed in the Sensor Abstract Layer (Queries 1 and 3).

6.4.1 Recurring Queries

A second approach for the application layer to derive information from the knowledge base is by *registering interest* in a recurring situation that is triggered by periodic timing events. Whenever such a situation occurs, a *Notify()* call returns a structured event that represents the predicate of interest to the application layer. Contrary to queries, recurring queries *do not examine the current state* of the Sentient World in order to establish whether the situation of interest holds at the current instance. Rather, they act similarly to a *subscribe* call in a *publish-subscribe* protocol for distributed systems, in registering interest in receiving information about *future* occurrences of the situation in question. Event-based asynchronous communication of changes in knowledge predicates is discussed in Chapter 8.

For example, an application may be interested in a regularly recurring event such as "Whenever any two people are co-located, update the GUI so that co-located people are portrayed as being enclosed in a rectangular area." A ***recurring query*** can be viewed as a first-order logical expression $f(c_1, c_2 \cdots c_n)$ that has the ***property*** that upon the satisfaction of a set of *atomic formulae* $c_1, c_2 \cdots c_n$, a *set of actions* are triggered.

$$f(c_1, c_2 \cdots c_n) \Rightarrow Notify(event)$$

The *Notify (event)* call passes on to the application layer a *structured event* that contains the queried information. Chapter 8 introduces abstract events as changes of abstract state. For example, such an event can be a "Supervisor Alert" event which is defined elsewhere in the system. When such an event is received by the application, the latter sends an appropriate e-mail message to the user. In fact, a particular case of recurring queries, is that upon satisfaction of the query, a notification action is performed. For example, "Whenever my supervisor enters the lab, notify me." Recurring queries can be expressed as *logical implications* where the left-hand-side is a simple query and the right-hand-side is a *Notify(event)* predicate.

Query 5. *Whenever my supervisor enters the lab, notify me by email.(Deductive Abstract Layer)*

$$uid|(H\_UserIsPresent(\text{uid Andy}) \Rightarrow \text{Notify(Supervisor Alert)})$$

The application layer, on receipt of the *Supervisor Alert* event, is responsible for issuing an appropriate e-mail notification. ***Note*** that this is equivalent to a high-level query, as it assumes that the predicate $H\_UserIsPresent$ is already available in the knowledge base.

6.5 Analysis

Having discussed queries and recurring queries, this section illustrates how the two-layer knowledge scheme ensures scalability. A prototype implementation was constructed, where queries and recurring queries are implemented in each layer by means of a CLIPS [19] inference engine. Each query is mapped to one or more CLIPS rules. CLIPS implements a forward chaining rule interpreter that, given a set of *rules* applied to a set of stored facts, cycles through a process of matching rules to available facts, thus determining which queries are satisfied by the stored state of the Sentient environment. The process by which CLIPS determines which facts satisfy the conditions of each query or recurring query is called *pattern matching*, and the *Rete algorithm* [38] is used for this purpose.

The advantage of the proposed architecture is due to three important factors:

- First, as can be seen from Sections 6.4 and 6.4.1, *queries* that are executed in the Deductive Abstract Layer such as Query 2, assume a *much simpler form* than those executed in the Sensor

Abstract Layer (Query 1), as the latter have *more conditions on average*, and therefore require *more computational resources for pattern matching*.

- Secondly, pattern matching is triggered *repeatedly* every time the stored knowledge changes by an *assert* or *retract* command. Therefore, the lower the rate of knowledge updates, the lower the computational load required (see figure 6.2). Since the knowledge update rate in DAL is significantly lower than the one in SAL (produced by the regular updates of the sensor infrastructure), *DAL is computationally more efficient*.

- Finally, it can be logically inferred from the above that as long as DAL is updated at a lower rate than SAL, the machine that hosts DAL has fewer real-time constraints that are introduced by the *interruptions* caused by the *assert* and *retract* statements that control knowledge updates.

The next session discusses the computational complexity associated with queries, in more detail, by analysing the Rete algorithm.

6.6 Prototype Implementation

This section aims to give a quantitative evaluation of the proposed scheme and its algorithm by discussing an implementation of the proposed system and by comparing Query 3 (see Section 6.4), which is executed at the Sensor Abstract Layer, to the same query (Query 4), which is executed at the Deductive Abstract Layer, and demonstrate that the latter entails a significantly smaller number of computational steps.

The proposed architecture was implemented using the Jess [51] production system. Jess is a java-based implementation of CLIPS. For the acquisition of real-time location information, a middleware component was built [46] that interfaces the Active BAT system using CORBA structured events and translates them into Jess facts.

6.6.1 Sensor Abstract Layer.

A model was created in Jess for the LCE based on location data produced by the Active BAT. The experiment involved 15 members of the lab moving around 21 known locations in the LCE. The following query "Return All Co-Located Users" was executed in the Sensor Abstract Layer.

Query 6. *Return All Co-Located Users (Sensor Abstract Layer).*

$(uid_1, uid_2)|$ (*L_UserInLocation*(uid $?uid_1$)(role $?role_1$)(rid $?rid$)(rattr $?rattr$))$\wedge$
(*L_UserInLocation*(uid $?uid_2$)(role $?role_2$)(rid $?rid$)(rattr $?rattr$))$\wedge$
(*L_AtomicLocation*(rid $?rid$)(rattr $?rattr$)(polygon $?n_1 ?n_2 \cdots ?n_j$))$\wedge$
$(uid_1 \neq uid_2)$

The Rete Algorithm. This implementation uses the Rete Algorithm [38] for pattern matching. In the Rete algorithm, the pattern compiler creates a network by linking together nodes that test query elements. This network functions similarly to a finite state machine whenever a query is added to the knowledge base, or whenever a new fact is asserted or retracted. Portrayed in salmon is the root node of the network, n_0. For each predicate included in the query, the network creates a one-input node, portrayed in *red* in Fig 6.3. Node n_2 corresponds to the predicate *L_AtomicLocation*. Node n_3 corresponds to the predicate

L_UserInLocation. Node n_4 is an auxiliary, *memory* node that helps implement a loop. A two-input *(green)* node is created for each conjunction of predicates. Node n_5 corresponds to the conjunction:

$$(\textit{L\_UserInLocation}(\text{uid } ?uid_1)(\text{role } ?role_1)(\text{rid } ?rid)(\text{rattr } ?rattr)) \wedge$$
$$(\textit{L\_UserInLocation}(\text{uid } ?uid_2)(\text{role } ?role_2)(\text{rid } ?rid)(\text{rattr } ?rattr))$$

Node n_6 corresponds to the conjunction:

$$(\textit{L\_UserInLocation}(\text{uid } ?uid_1)(\text{role } ?role_1)(\text{rid } ?rid)(\text{rattr } ?rattr))\wedge$$
$$(\textit{L\_UserInLocation}(\text{uid } ?uid_2)(\text{role } ?role_2)(\text{rid } ?rid)(\text{rattr } ?rattr))\wedge$$
$$(\textit{L\_AtomicLocation}(\text{rid } ?rid)(\text{rattr } ?rattr)(\text{polygon } ?n_1 \cdots ?n_j))$$

Node n_1 is an auxiliary node that helps implement node n_7 that represents the condition $(uid_1 \neq uid_2)$. Finally, node n_8 is a terminal node that determines whether the query is satisfied or not.

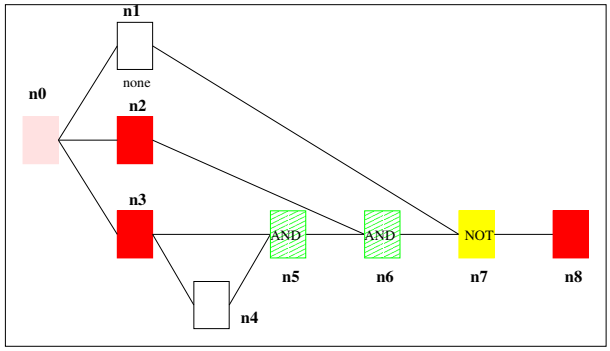


Figure 6.3. The Rete network for the Return All Co-Located Users query (SAL)

The Rete algorithm proceeds as follows: when the query is added to the Sensor Abstract Layer, for each stored fact, a *token* is created. Each token is an ordered pair of a tag (that in this case has the value "UPDATE") and a description of the stored fact. All these tokens are passed to node n_0, which is the root node in the network. Node n_0 passes all the generated tokens to each of its successor nodes. Node n_3 checks whether any of the received tokens correspond to facts of type *L_UserAtLocation*[6] and passes all such tokens to node n_4. Node n_5 checks all *L_UserAtLocation* tokens against each other, in order to determine which pairs satisfy the conjunction:

$$(\textit{L\_UserInLocation}(\text{uid } ?uid_1)(\text{role } ?role)(\text{rid } ?rid)(\text{rattr } ?rattr) \wedge$$
$$(\textit{L\_UserInLocation}(\text{uid } ?uid_2)(\text{role } ?role)(\text{rid } ?rid)(\text{rattr } ?rattr))$$

For each of the pairs that satisfy the conjunction, it creates a new token and forwards this on to node n_6. Node n_2 tests for tokens that are of type $L\_AtomicLocation$ and passes these on to node n_6, too. Node n_6 joins the pairs that represent the conjunction:

$$(\textit{L\_UserInLocation}(\text{uid } ?uid_1)(\text{role } ?role_1)(\text{rid } ?rid)(\text{rattr } ?rattr)) \wedge$$
$$(\textit{L\_UserInLocation}(\text{uid } ?uid_2)(\text{role } ?role_2)(\text{rid } ?rid)(\text{rattr } ?rattr)) \wedge$$
$$(\textit{L\_AtomicLocation}(\text{rid } ?rid)(\text{rattr } ?rattr)(\text{polygon } ?n_1 \cdots ?n_j))$$

[6] In this prototype implementation, the SPIRIT system was used to provide *L_UserAtPosition* predicates from the Active BAT positions.

	Sensor Abstract Layer (SAL)	Deductive Abstract Layer (DAL)
total node activations	317	200
total tests on nodes	1611	0

Table 6.2. Pattern matching costs.

into bigger tokens and forwards them on to n_7. Node n_7 tests that $uid_1 \neq uid_2$ thus excluding trivial co-locations of the same person. It forwards the eligible tokens to n_8, the success node. These tokens satisfy the whole query. For each token, n_8 creates an instantiation of the query.

In order to obtain a measure of the computational complexity that the implemented scheme entails, the number of node activations and the number of tests performed in total by the nodes on the network was investigated. Both Queries 3 and 4 were asserted in the prototype implementation of Chapter 10. The results are shown in Table 6.2 (SAL).

6.6.2 Deductive Abstract Layer

The previous experiment was repeated for Query 4 (see Section 6.4) which is executed in the Deductive Abstract Layer. The network for this query is portrayed in Figure 6.4.

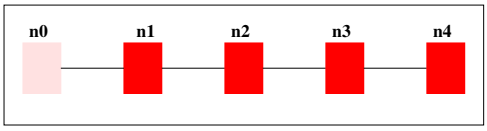


Figure 6.4. The Rete network for the Return All Co-Located Users query (DAL)

Node n_0 is the root node. Node n_1 tests whether the received token is of type *H_UserCoLocation*. Node n_2 passes on the tokens with the correct number of arguments and n_4 creates an instantiation of the query and adds it to the conflict set.

Performing the same analysis as before, the results are presented in Table 6.2. It is worth noting that the number of computational steps executed by the Rete algorithm for pattern matching each query is lower for the DAL query by a factor of two. Taking into consideration that both networks (see Figure 6.3, Figure 6.4) behave similarly to acyclic finite automata that are triggered repeatedly each time a fact is asserted or retracted in each knowledge layer respectively, it can be easily inferred that the overall number of computational steps required for DAL is smaller than that required for SAL as knowledge in DAL changes much less frequently. Finally, SAL is continually *interrupted* by a very high event rate which has an immediate effect on the machine that hosts that layer.

6.7 Conclusions

A scalable knowledge representation and abstract reasoning system for Sentient Computing was presented where knowledge was modelled formally using first-order logic. First-order logic is suitable for Sentient Computing, especially in the context of the proposed architecture that is based on a cache-like, dual-layer scheme that maintains abstract knowledge in the Deductive Abstract Layer as opposed to rapidly changing low-level knowledge in the lower, Sensor Abstract layer. Abstract knowledge remains consistent with the rapidly changing state of the Sentient world by closely monitoring associated, low-level predicates as requested by the application layer through an API. Such predicates are contained in the Sensor Abstract Layer and by having only threshold changes reflected at DAL. Maintaining abstract knowledge is a requirement of the Sentient Application layer, and it is made available to Sentient

Applications through a mechanism of *queries* that are mainly executed at the Deductive Abstract layer. Experiments with a prototype implementation (see Section 6.6) confirm that the two-layered architecture is more efficient than a single-layered one.

Chapter 7

Query Analysis and Optimisation

This chapter describes a worst-case analysis of how the computational efficiency of the previously described model is affected whenever a new query is inserted in the knowledge base. It also analyses the effect of the event rate on the computational efficiency of SCAFOS' deductive knowledge base component for the same worst-case scenario. The findings of this analysis are combined in order to optimise queries. *Query optimisation* has an immediate effect on the design of the AESL language presented in Chapter 12, as its statements are mapped onto optimised queries.

7.1 The Effect of Query Assertion on the Computational Complexity

Having established in Chapter 6 that pattern matching is key to the computational load involved in executing a query, this section introduces a complexity metric for queries based on the concept of query *response time*.

Def. 1. *The response time of a query is the time that passes from the moment a query is asserted until an answer is available.*

The response time for any query can be calculated in terms of the number of *node activations* and the number of *tests* that are performed in the nodes of the Rete network that implements that query. Table 7.1 describes the most important metrics of interest. Every time a sensor creates a new instance of a concrete state predicate, corresponding tokens are created and propagated through the arcs to the nodes.

T_q	Is the total time required to assert a query.
N_1	Is the number of activations of one-input nodes.
a	Is the cost of activating a one-input node (including the cost of the test performed by the node).
N_2	Is the number of activations of two-input nodes, including the number of activations of memory nodes and the number of updates performed in the memory nodes.
γ	Is the combined cost of activating a two-input node, the cost of activating the two memory nodes that form its right and left memory and the cost of performing a test on these memory nodes.
T_2	Is the number of tests performed by the two-input nodes.
c	Is the cost of performing one test at a two-input node.
N_p	Is the number of activations of $\&P$ nodes.
f	Is the cost of activating a $\&P$ node.

Table 7.1. Cost analysis of the Rete algorithm.

Node Types

Five types of Rete network nodes are discussed in this chapter. *One-input* nodes, *two-input* nodes, *memory* nodes, $\&P$ nodes and TQ nodes[1]. Two more node types are discussed in Section 8.3.1, *Store* nodes and *NOT* nodes.

Each *one-input* node checks whether the received tokens correspond to a particular condition, e.g., if they are of class $H\_UserInLocation$. These nodes are portrayed in red. One-input nodes also check whether a value is assigned correctly to an attribute. Such nodes are portrayed in brown and they are discussed in more detail in Chapter 12. Each one-input node forwards the tokens that satisfy the check on to its child nodes.

Two-input nodes represent conjunctions. They contain two memory nodes called the *left* and *right* memory node. They concatenate the tokens that are stored in their right and left memory and they perform a test to determine whether shared variables are bound correctly. Such nodes are portrayed in green and they are labelled "AND".

Trigger-Query (TQ) nodes are nodes that trigger a JESS query that selects all instances of a particular predicate from the knowledge base, for each token that is received at that node. Each (TQ) node is portrayed as a pair of identical nodes connected with a curvy line. TQ nodes are integral to the implementation of Rete Networks that implement functions such as those that calculate the maximum or minimum value of an attribute of all stored instances of a predicate. They are often used in this dissertation for calculating the location with the smallest distance to one of the users. Each of the two nodes that form a TQ node is labelled "TQ (predicate)".

Finally, $\&P$ nodes are final nodes. There is one $\&P$ node for each query. When a token is forwarded to the final node, an instance of the abstract predicate that is being defined is created or deleted accordingly.

Table 7.1 describes some metrics of interest. According to [37], the response time of a query can be estimated as:

$$T_q = aN_1 + \gamma N_2 + cT_2 + fN_p. \tag{7.1}$$

Sensor Abstract Layer: Worst-Case Scenario

Let's assume a query that selects any n users that are contained within m regions. The query is executed at the Sensor Abstract Layer, which contains N facts in total. Out of these N facts, n facts are of type $L\_UserAtPosition$, m facts are of type $L\_AtomicLocation$ and n facts are of type $L\_InRegion$, because we assume that for each $L\_UserAtPosition$ predicate there is an equivalent $L\_InRegion$ fact, which associates this position with a region. Therefore, $N = m + 2n$. The worst case query is one where each condition introduces as few constraints as possible, while a large number of different conditions are conjoined. The following query (Query 7) has $C_1 = n$ conditions about facts of type $L\_UserAtPosition$, $C_2 = m$ conditions about facts of type $L\_AtomicLocation$ and $C_3 = n$ conditions about facts of type $L\_InRegion$, all conjoined.

[1] The nomenclature and notation for these nodes is the one found in [37].

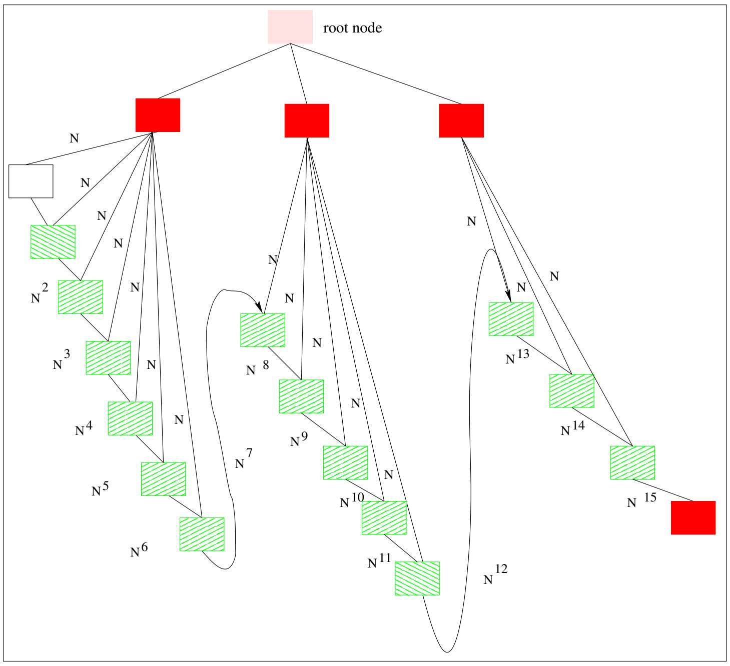


Figure 7.1. A worst-case query (SAL).

Query 7. *A Worst Case Query (Sensor Abstract Layer).*

$$
\begin{aligned}
& H\_UserInLocation(uid_1, role_1, rid_1, rattr_1) \wedge \\
& H\_UserInLocation(uid_2, role_2, rid_2, rattr_2) \\
& \vdots \\
& H\_UserInLocation(uid_n, role_n, rid_m, rattr_m) \wedge \\
& L\_AtomicLocation(rid_1, rattr_1) \wedge \\
& \vdots \\
& L\_AtomicLocation(rid_m, rattr_m) \wedge \\
& L\_InRegion(x_1, y_1, z_1, rid_1) \wedge \\
& \vdots \\
& L\_InRegion(x_n, y_n, z_n, rid_m) \wedge
\end{aligned}
$$

Figure 7.1 depicts the Rete network for the above query where 7 users are located inside 3 out of 5 rooms[2] $C_1 = 7$, $C_2 = 5$, $C_3 = 3$, $C = 15$. In this query, each one-input node passes all the tokens that activated it to its successors. In each case, the number of the tokens that have activated the one-input node is smaller than N, so N is the worst-case value for number of activating tokens for each one-input node. Each two-input node at level k receives a maximum of N tokens from the right input and N^{k-1} tokens from the left input and performs $N \cdot N^{k-1}$ tests. In the worst case, all these tests will be successful, and they will produce N^k new tokens. Each two-input node maintains two memory nodes, which correspond to the right and left memory for that node. There are $C - 1$ levels of two-input nodes

[2] It is assumed that the rest of the $7 - 3 = 4$ positions cannot be mapped into any room in the environment.

where $C = C_1 + C_2 + C_3$ is the total number of conditions in the query.

The total number of node activations and node tests are given by the following equations.

$$\begin{aligned} N_1 &= 3N && (7.2)\\ N_2 &= 2N + N + N^2 + N + N^3 + N + \cdots + N^{C-1} + N \\ &= CN + N^2 + \cdots + N^{C-1} && (7.3)\\ N_p &= N^C && (7.4)\\ T_2 &= N^2 + N^3 + \cdots + N^C && (7.5)\\ & && (7.6) \end{aligned}$$

Therefore, it follows from Equation 7.1:

$$\begin{aligned} T_q &= (3a + \gamma C)\, N + (\gamma + c)N^2 + \cdots + (\gamma + c)N^3 + \cdots + (\gamma + c)N^{C-1} + (\gamma + f)N^C \\ &= O\left(N^C\right) && (7.7) \end{aligned}$$

Deductive Abstract Layer: Worst-Case Scenario.

The equivalent, worst case, single query for the Deductive Abstract Layer would have one condition that depends on the high-level predicate of interest. The previous query in the Deductive Abstract Layer would be represented by the predicate $H\_n\_Users\_In\_m\_Locations$ *(nUImL)*:

Query 8. *The H_n_Users_In_m_Locations Query (Deductive Abstract Layer).*

$$H\_n\_Users\_In\_m\_Locations(uid_1, \ldots, uid_n, role_1, \ldots, role_n, rid_1, \ldots, rid_m, rattr_1, \ldots, rattr_m)$$

The variables uid_1 to uid_n represent the n users with roles $role_1$ to $role_n$ that are contained in the m locations rid_1 to rid_m with attributes $rattr_1$ to $rattr_m$. Figure 7.2 portrays the Rete network that corresponds to the above query without constants. An estimate for the maximum number of computational steps involved in this case can be calculated from Equation 7.1, noting that because of the lack of any two-input nodes, $N_2 = T_2 = 0$.

$$\begin{aligned} T_q &= N + fN \\ &= (1 + f)N \\ &= O(N) && (7.8) \end{aligned}$$

7.1.1 The Effect of the Event Rate on the Computational Complexity of the Worst-Case Scenario Query

This section investigates the computational load involved in the reception of an event that causes the queries that are already stored in the knowledge base to be re-evaluated against the modified set of facts in the knowledge base. In order to introduce an estimate for the maximum number of computational steps that are triggered from receiving an event, the pattern matching process is investigated and analysed next.

Assuming that Query 7 has already been matched against the previous N elements, the contents of the memory of the nodes are depicted in Figure 7.3. Upon the assertion of each *L_UserAtPosition* fact, the new fact activates all three one-input nodes, but only the left one, which tests whether it is of type $L\_UserAtPosition$ will pass it on. The first two-input node will test the new fact against its right

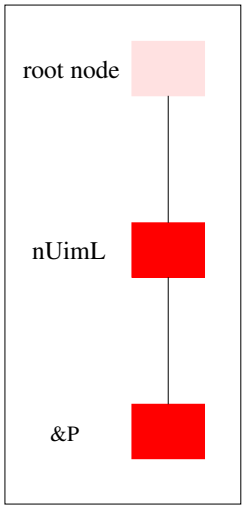


Figure 7.2. A worst-case query (DAL).

memory, which already contains $N+1$ facts, and it will, therefore, perform $N+1$ tests. It will also pass $N+1$ tokens in the worst case to its successor node. This node will perform $(N+1)(N+1) = (N+1)^2$ tests, and it will pass $(N+1)^2$ tokens on. Without affecting the computational complexity, $N+1$ can be approximated by N. Because the reception of a new *L_UserAtPosition* fact entails the retraction of its old instance, the retracted fact is also pattern-matched against the Rete network in order to determine which rules no longer hold, and to remove their instantiations from the conflict set. Therefore, in total, for each new $L\_UserAtPosition$ fact, two traversals of the Rete network are required and, as follows from Equation 7.1, the total time cost that is incurred by the reception of each event is T_f, as can be derived from the equation below:

$$\begin{aligned} T_f &= 2T_q \\ &= 2[aN1 + \gamma N_2 + cT_2 + fN_p] \end{aligned} \tag{7.9}$$

where T_q is the time required for the pattern matching of one fact.

Sensor Abstract Layer: Worst-Case Scenario

$$N_1 = 3 \tag{7.10}$$

$$\begin{aligned} N_2 &= 2 + N + 1 + N^2 + 1 + \cdots + N^{C-2} \\ &= N + N^2 + \cdots + N^{C-2} + C_1 \end{aligned} \tag{7.11}$$

$$N_p = N^{C-1} \tag{7.12}$$

$$T_2 = N + N^2 + N^3 + \cdots + N^{C-1} \tag{7.13}$$

Taking into account that events are received and asserted in the Sensor Abstract Layer as facts with rate b, then it follows from Equation 7.9:

$$\begin{aligned} T &= bT_f \\ &= 2b\left(a + \gamma N + (\gamma + c)\, N^2 + (\gamma + c)\, N^{C-2} + \gamma C_1 + (c + f)\, N^{C-1}\right) \\ &= O\left(N^{C-1}\right) \end{aligned} \tag{7.14}$$

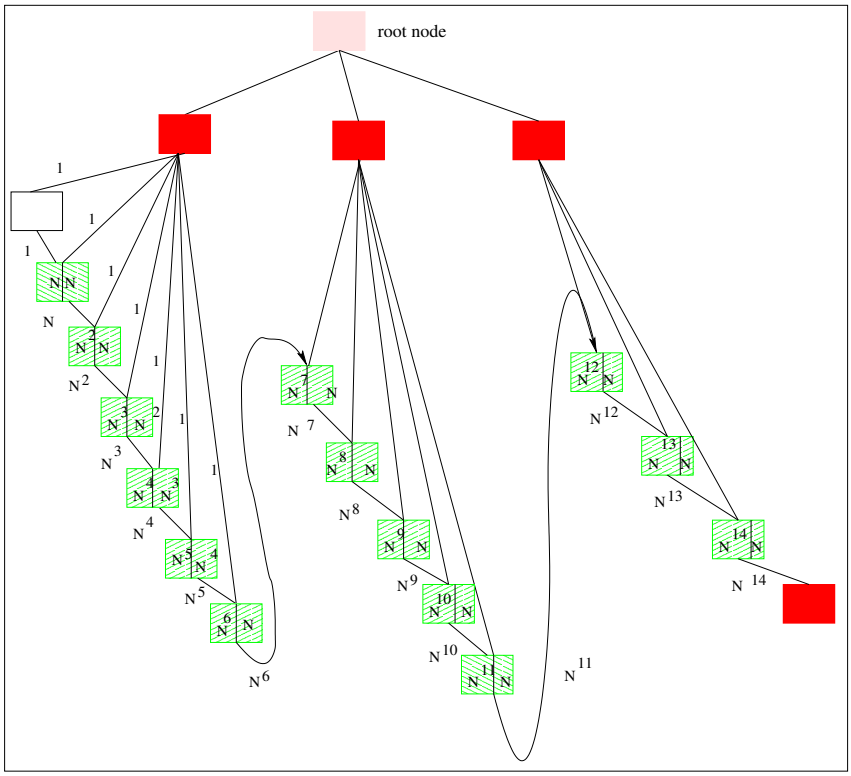


Figure 7.3. The effect of the event rate on the worst-case query of Figure 7.1.

As can be seen from the above, the computational complexity involved in each event reception depends on the rate with which the predicate changes, the size of the Sentient world (number of monitored people) as well as the architecture of the query in terms of the number of nodes to which it compiles.

Deductive Abstract Layer

Let b' be the rate with which instances of the predicate $nUImL$ change in DAL. Calculating the event rate in the network of Figure 7.2 when a fact of type $L\_UserAtPosition$ is asserted in the knowledge base:

$$\begin{aligned} N_1 &= N + N \\ N_2 &= 0 \\ T_2 &= 0 \\ N_p &= N \end{aligned} \tag{7.15}$$

From Equation 7.9, it follows that $T = O(N)$.

7.2 Query Optimisation

The above analysis is instrumental in determining the nature of the Rete networks into which AESL definitions of Chapters 8 and 12 are compiled. In fact, the design of the AESL definitions should be such

that the worst-case Rete network is avoided. Chapter 12 discusses the syntax of AESL definitions, here referred to for simplicity as *rules* (see Section 1.8).

The worst-case Rete network for a given rule occurs by conjoining all the conditions. Instead, conditions can be written as separate implications that result in intermediate abstract facts being asserted in SAL. We demonstrate that Rule 9 is computationally more efficient than Rule 10.

Rule 9. *Locate the closest, non-empty room to each user.*

$$
\begin{aligned}
& H\_UserInLocation(uid, rid, role, rattr) \\
\Rightarrow\ & H\_NonEmptyLocation(rid, rattr) \\
& H\_Distance(v_1, uid, role, rid_2, rattr_2) > H\_Distance(v_2, uid, role, rid_1, rattr_1) \\
\Rightarrow\ & H\_ClosestLocation(uid, role, rid_1, rattr_1) \\
& H\_ClosestLocation(uid, role, rid, rattr) \ \wedge H\_NonEmptyLocation(rid, rattr) \\
\Rightarrow\ & H\_ClosestNonEmptyLocation(uid, role, rid, rattr)
\end{aligned}
$$

Rule 10. *Locate the closest, non-empty room to each user.*

$$
\begin{aligned}
& H\_UserInLocation(uid_k), rid_1, role, rattr_1) \\
& \wedge H\_Distance(v_1, uid, role, rid_2, rattr_2) > H\_Distance(v_2, uid, role, rid_1, rattr_1) \\
\Rightarrow\ & H\_ClosestNonEmptyLocation(uid, role, rid_1, rattr)
\end{aligned}
$$

The Rete networks for the above rules can be seen in Figure 7.4 and Figure 7.5, respectively. $CNEL$ stands for $H\_ClosestNonEmptyLocation$, NEL stands for $H\_NonEmptyLocation$ and CL for $H\_ClosestLocation$. Each token of type $D(v, uid, role, rid, rattr)$ that is received by the node $TQ(D(v, uid, role, rid, rattr))$ is tested against all tokens of the same type that are stored in the knowledge base. For example, node $TQ(D(v, uid, role, rid, rattr))$ receives mn tokens of type $D(v, uid, role, rid, rattr)$ and performs (m^2n^2) tests.

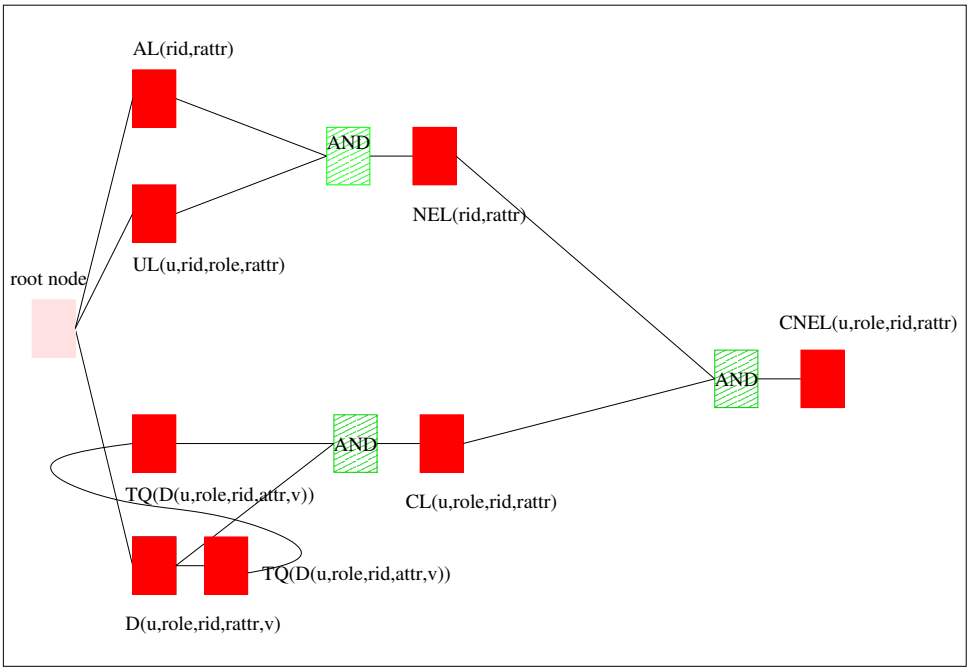


Figure 7.4. Implications.

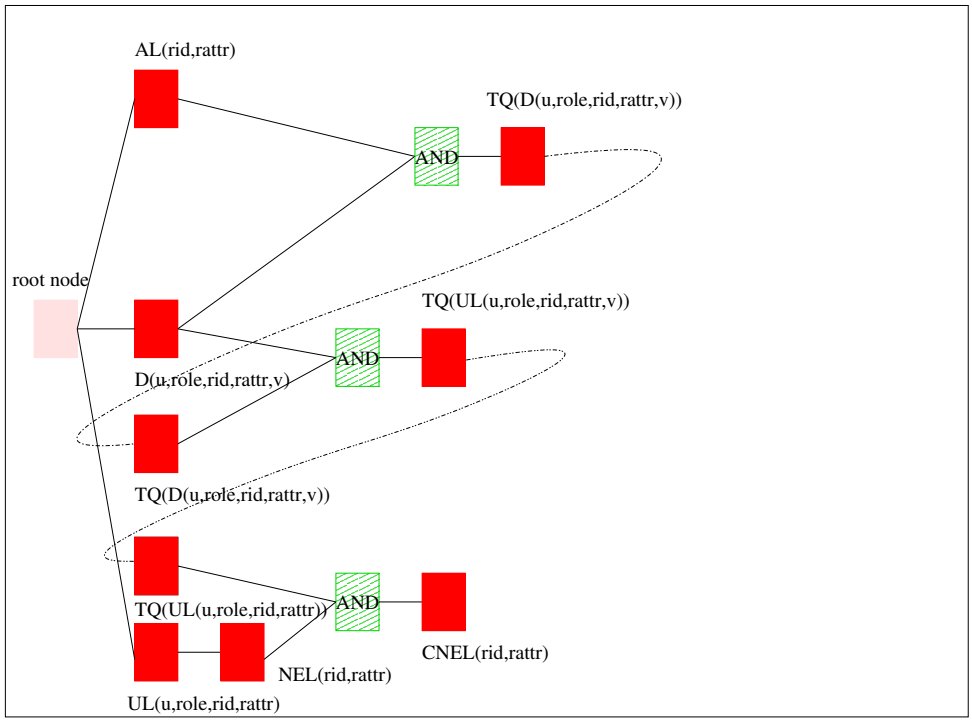


Figure 7.5. Conjunction.

7.2.1 Analysis of Rule 9

We assume n $H\_UserInLocation$ facts, m $L\_AtomicLocation$ facts, nm $H\_Distance$ facts.

$$\begin{aligned} N_1 &= 3N + mn + mn + n + n \\ N_2 &= m + n + 2mn + m + n \\ T_2 &= mn + m^2n^2 + mn \\ N_p &= m \end{aligned}$$

The computational complexity from asserting this rule into the knowledge base is given by Equation 7.1. Because the total number of stored facts in the knowledge base is N, the total cost amounts to $O(N^4)$.

7.2.2 Analysis of Rule 10

$$\begin{aligned} N_1 &= 3N \\ N_2 &= mn + mn + m^3n^2 + mn + m^4n^+m \\ T_2 &= m^2n + m^4n^3 + m^5n^4 \\ N_p &= N \end{aligned}$$

The computational complexity from asserting this rule into the knowledge base is given by Equation 7.1 and it amounts to $O(N^9)$.

7.2.3 Conclusions and Further Work

- From Equations 7.7 and 7.14 it follows that, for worst-case queries, the computational complexity that is involved in their evaluation is N^C every time the query is asserted or N^{C-1} everytime a new primitive event is received. C denotes the number of conditions of the query and N denotes the number of facts in the knowledge base.
- Worst-case Rete networks are those that correspond to rules that are conjunctions of conditions. The above complexity is significantly reduced when rules are written as a set of implications that are correlated by the inference engine. This advocates that the design of AESL Definitions should contain implications.
- Care needs to be taken to prevent the number of facts that are generated by the implications of AESL definitions from becoming too large.

Chapter 8

An Extended Publish/Subscribe Protocol Using Abstract Events

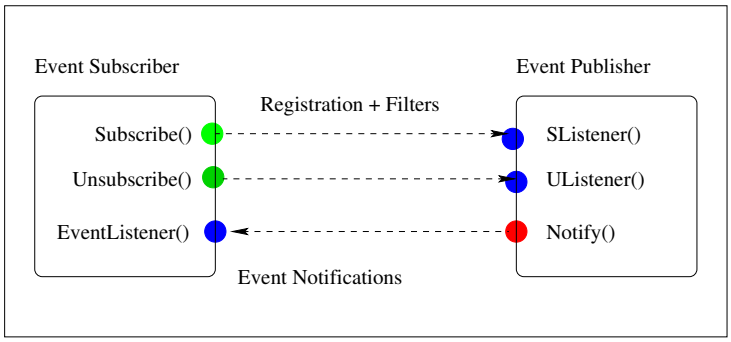
Most distributed systems research assumes that events are primitive, and various studies have, therefore, concentrated on *composite events*. However, Chapter 5 demonstrated that event-based systems, such as those using finite state machines, are insufficient for querying and subscribing transparently to distributed state. This is due to the fact that the mapping between the subscription language and the implementation domain is incomplete, which makes computation by finite automata limited. This necessitates an alternative model for ubiquitous sensor-driven systems.

This chapter proposes the notion of an *abstract event* as a notification of transparent changes in distributed state. This is implemented as an extension to the publish/subscribe protocol in which a higher-order service (Abstract Event Detection Service) publishes its interface; this service takes an abstract event specification as an argument and in return publishes an interface to a further service (an abstract event detector), which notifies transitions between the values *true* and *false* of the formula, thus providing a natural interface to applications.

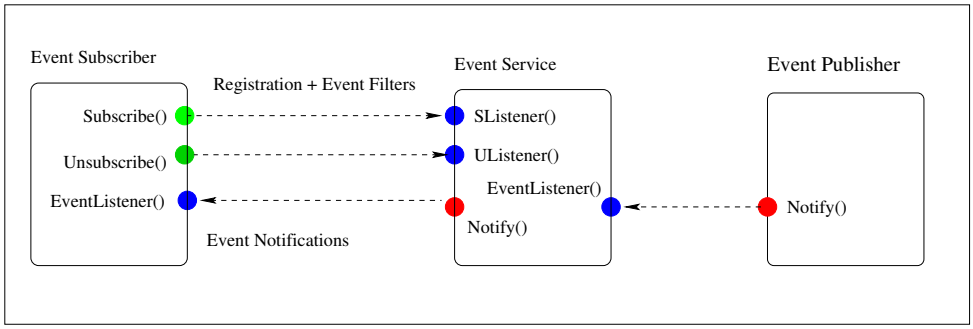
8.1 The Publish/Subscribe Protocol

Contemporary large-scale distributed systems tend to be designed as assemblies of *loosely coupled components* communicating by means of the publish/subscribe model for event interaction. The publish/subscribe event-interaction scheme is widely used in order to provide the loosely coupled form of interaction required in such large-scale settings. It provides a subscriber with the ability to express their interest in an event or a pattern of events in order to be notified afterwards of any event, generated by a publisher, matching the registered interest.

In the basic system model for publish/subscribe interaction, the publisher publishes its interface (SListener), including the events it will notify. A subscriber *registers* interest in events indicating, where appropriate, constraints on the event parameters. The publisher *notifies* the subscriber of event occurrences that match the subscriber's registration. An event service, such as the CORBA notification service, can act as a mediator between the publisher and the subscriber decoupling the subscriber and the publisher in space, flow and time, undertaking event filtering and event storage and, at the same time, providing services such as message buffering and message forwarding to disconnected subscribers [7]. In this scheme, subscribers register their interest in events by typically calling a *Subscribe()* operation on the event service without knowing the publishers of these events. A symmetric operation *Unsubscribe()* terminates a subscription. To generate an event, a publisher calls a *Notify()* operation on the event service. The event service directs the call to all relevant subscribers so that *every subscriber* receives a notification for every event conforming to its registration.



(a)



(b)

Figure 8.1. The publish/subscribe protocol. Direct publisher-subscriber interaction (a). Publisher-subscriber interaction through an Event Service (b).

8.2 An Abstract Event Model

In this section, the model of Chapter 5 is summarised in order to provide a definition for abstract events. It is assumed that a system consists of several physical *domains* such as an office domain. Each domain contains a set of physical objects, such as users and equipment, that are mobile and can move freely in space. Each object has a *state* which is implemented in our model through a list of *state predicates* that can be either true or false. State predicates can be either concrete or abstract. Concrete state predicates represent state that is directly derived from the sensors; in this dissertation, such predicates are always prefixed with "*L_*"(Low-level). Examples of concrete state predicates are the states that represent a user's position in terms of his coordinates, rooms in a building and nested locations such as floors. These are modelled with the first-order logic (FOL) predicates $L\_UserAtPosition$, $L\_AtomicLocation$ and $L\_NestedLocation$, respectively. For example, *L_UserAtPosition(John, 13.45, 5.76, 1.75, 13:56)*, *AtomicLocation(Room-7, Meeting Room, Polygon)*, *L_NestedLocation(Floor4, Room 1,Room 2, Room 3, Room 4)*.

Abstract state predicates represent high-level state that is derived from concrete state by means of abstractions on properties of interest; for example, a user's high-level location in terms of the region that contains that user, a user's presence or absence and the fact that one or more people are co-located. Initially, when the system is started up, only concrete state predicates exist. Abstract state predicates in this paper are always prefixed with "*H_*"(High-level).

Definition 1. *An abstract event is detected when an instance of an abstract state predicate which initially evaluates to true next evaluates to false and vice versa.*

8.3 Abstract Event Specification and Filtering

The *Abstract Event Specification Language (AESL)*[1] for creating abstract event definitions is a subset of TFOL that corresponds to Horn Clause Logic. *An abstract event definition* (AESL def) consists of one or more *implications* (Horn clauses). In case of only one implication, the RHS is the abstract predicate of interest. In case of more than one implication, the RHS of the last rule is the abstract predicate of interest while the RHS of each intermediate rule is an intermediate abstract predicate.[2] The AESL language is discussed in detail in Chapter 12. An example is given here for illustration purposes.

Example 1. *Locate the closest location to each user.*

Writing for brevity UL for $H\_UserInLocation$, AL for $L\_AtomicLocation$, EL for $H\_EmptyLocation$, CL for $H\_ClosestLocation$, CEL for $H\_ClosestEmptyLocation$ and D for $H\_Distance$:

$$
\begin{aligned}
&(\nexists u\ UL(u, rid, role, rattr) \wedge AL(rid, rattr, polygon) \\
&\quad \Rightarrow EL(rid, rattr)) \\
&D(v_1, u, role, rid_2, rattr_2) > D(v_2, u, role, rid_1, rattr_1) \\
&\quad \Rightarrow CL(u, role, rid_1, rattr_1) \\
&CL(u, role, rid, rattr) \wedge EL(rid, rattr) \\
&\quad \Rightarrow CEL(u, role, rid, rattr)
\end{aligned} \tag{8.1}
$$

[1] *AESL is a typed language, however types are ignored in this chapter. The AESL language is discussed extensively in Chapter 12.*

[2] As usual with these clauses, all free variables are presumed to be implicitly universally quantified.

Filtering. Horn Clause logic is used for defining filters during client subscription. Filtering is equivalent to selecting a subset of instances of a specific predicate by specifying a set of constraints on its attributes. A filter that selects only the instances in (8.1) that correspond to meeting rooms and system administrators is shown in (8.2). Note that we encourage AESL definitions to use variables as arguments for predicates rather than constants. This facilitates implementation optimisation, and it ensures that the deduction of the abstract predicate, which is computationally expensive, is performed once, and multiple instances of the predicate are subsequently selected using filters. Section 8.3.1 discusses the implementation of AESL definitions and filters in more detail. The syntax of the AEFSL language that is used in SCAFOS for filtering is discussed in detail in Chapter 12.

$$(rattr = \textit{Meeting Room}) \wedge (role = \textit{Sysadm}) \tag{8.2}$$

8.3.1 Abstract Event Detectors

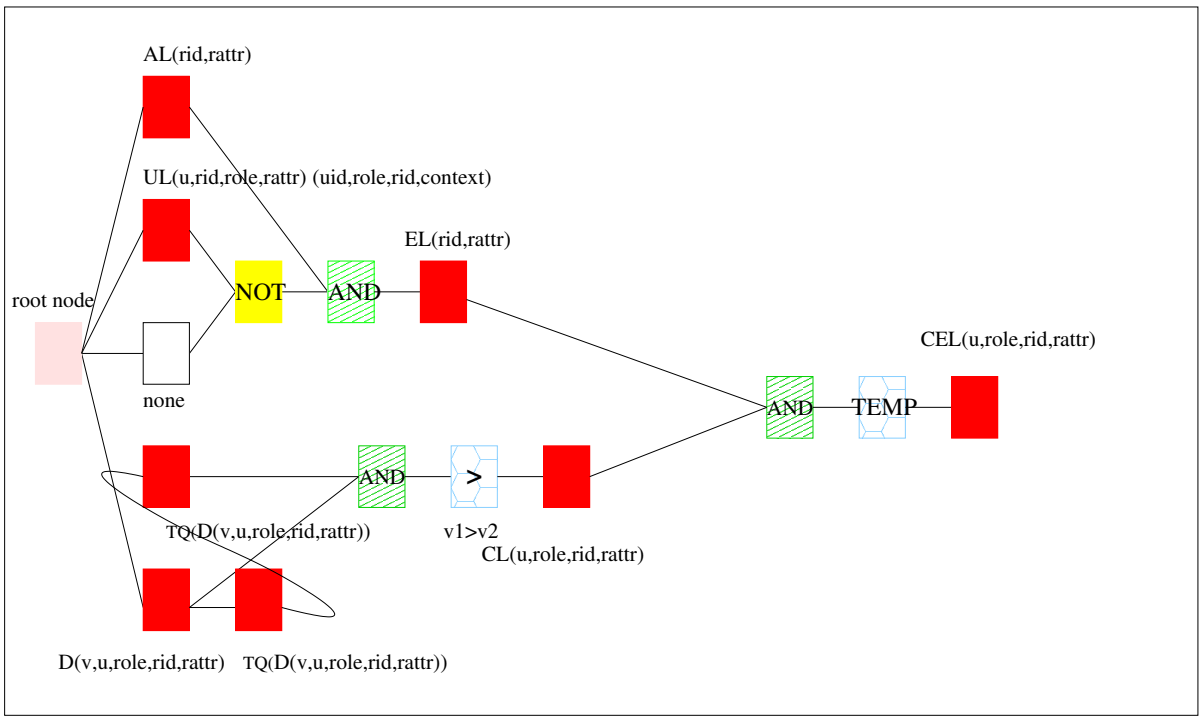


Figure 8.2. An abstract event detector for Equation (8.1).

Each AESL definition is compiled into one or more abstract event (AE) detectors that are structured as a deductive knowledge base, and that can perform semantic operations on instances of knowledge predicates that are defined in terms of TFOL formulae. They are implemented as Rete networks [38], and they consist of nodes and arcs. Every time a sensor creates a new instance of a concrete state predicate, corresponding tokens are created and propagated through the arcs to the nodes, eventually modifying appropriately the value of the abstract predicate.

Node Types

This section outlines the type of nodes that are found in Rete Networks. It extends the definitions of Section 7.1 by two additional node types - i.e., *Store* nodes and *NOT* nodes.

One-input nodes check whether the received tokens correspond to a particular condition, e.g., if they are of class $H\_UserInLocation$. These nodes are portrayed in red. One-input nodes also check whether a value is assigned correctly to an attribute. Such nodes are portrayed in brown, and they are discussed in more detail in Chapter 12. Each one-input node forwards the tokens that satisfy the check on to its child nodes.

Two-input nodes represent conjunctions. They concatenate the tokens that are stored in their right and left memory, and they perform a test to determine whether shared variables are bound correctly. Such nodes are portrayed in green and are labelled "AND".

Store nodes act as buffers for the current and historical instances of a predicate type and forward all stored instances on to the child nodes. This allows for temporal reasoning.

Trigger-Query (TQ) nodes are nodes that trigger a CLIPS query that selects all instances of a particular predicate from the knowledge base for each token that is received at that node. Each (TQ) node is portrayed as a pair of identical nodes connected with a curvy line. TQ nodes are integral in Rete Networks that implement functions such as those that calculate the maximum or minimum value of an attribute of all stored instances of a predicate. They are often used in this dissertation for calculating the location with the smallest distance to one of the users. Each of the two nodes that form a TQ node is labelled "TQ(predicate)".

NOT nodes are satisfied when there is no token in their right memory. They are two-input nodes that use a special, auxiliary token (*"none"*) in their left memory.

Test nodes perform a mathematical or logical operation such as equality or inequality on the values of the attributes of the tokens they receive.

Finally, $\&P$ nodes are final nodes. When a token is forwarded to the final node, an instance of the abstract predicate that is being defined is created or deleted accordingly and an "activation" or "de-activation" abstract event is triggered, respectively. An abstract event detector for (8.1) is portrayed in Figure 8.2. Each match in the network will cause the detection of the following abstract event:

$$\langle H\_ClosestEmptyLocation\langle uid, role, rid, \textbf{\textit{Meeting Room}}, activation, timestamp\rangle\rangle$$

Each time an instance of the abstract predicate ***H_ClosestEmptyLocation(uid, role, rid, Meeting Room)*** that was previously true is evaluated to false, the following event will be detected:

$$\langle H\_ClosestEmptyLocation\langle uid, role, rid, \textbf{\textit{Meeting Room}}, de\text{-}activation, timestamp\rangle\rangle$$

Abstract event detection can be distributed so that each implication in an AESL definition is implemented by a separate detector, forming a hierarchical topology similar to SIENA [15] and HERMES [78]. The optimal placement of the detectors in the system can be determined as appropriate [81].

A filter is implemented as an AE detector with linear complexity. Filters can be combined whenever there is a shared condition. For example, the filter of (8.2) can be combined with the filter of (8.3) as shown in Figure 8.3.

$$(rattr = Meeting\ Room) \wedge (role = Ceo) \tag{8.3}$$

8.3.2 Properties of Abstract Event Detectors

Negation. The proposed Rete-network-based implementation handles negation efficiently by using the deductive knowledge base component in order to generate the missing negated abstract predicates from concrete predicates, whenever this is possible. This is achieved by forcing concrete predicates that are negated to be *retracted* from the knowledge base. Indeed, for each fact asserted by a sensor, the previous instance of the same predicate is retracted from the knowledge base. If this predicate is modelled with the symbol P, the retraction is equivalent to the assertion of an instance of the $\neg P$ predicate. The change caused by the $\neg P$ predicate is propagated towards the abstract instances by means of the Rete networks,

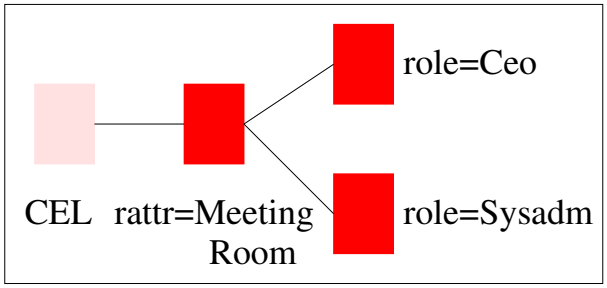


Figure 8.3. Filter combination.

and all abstract predicates that depend on P are forced to be re-evaluated. The abstract (deduced) predicates Q_i that no longer hold, as a result of the propagation of $\neg P$, are in turn retracted, which causes the assertion of the negative predicates $\neg Q_i$ to be deduced. In this way, the mapping between the FOL expressions and the domain becomes complete. This ensures that negation in FOL expressions with state is handled sufficiently.

Partial Knowledge. Because all asserted facts are maintained in the knowledge base, Rete networks have access to all of the available knowledge. So partial knowledge in the sense of finite-state machine implementations (see Chapter 5) does not exist here.

Dynamic Extensibility. One of the key advantages of this approach is that facts can be asserted and retracted from the system dynamically, and the set of deduced facts that represents abstract, deduced state is kept consistent with the change. This is due to the fact that each assertion or retraction is propagated to the set of deduced facts by the Rete networks, and all deduced facts are re-evaluated accordingly.

Maintaining Transparency. The key enabler of CAAT (see Section 5.2) is that Rete networks are higher-order and deal with free variables by default. Each node in the Rete network operates according to an algorithm that undertakes the testing of all relevant facts in memory that satisfy the node variables and its conditions. Therefore, the same Rete network can be used for the evaluation of the same expression in any domain, as long as the naming of predicate types is standard.

8.4 The Extended Publish/Subscribe Protocol

The proposed extension to the publish/subscribe protocol is equivalent to a *high-order service* (which is called Abstract Event Detection (AED) Service) where subscribers do not just subscribe to event notifications as in the traditional form of this protocol, but to the establishment and configuration of an abstract event detector (Section 8.3.1) for a new abstract event of interest. The AED Service acts as a mediator between the subscriber and the publisher, and is responsible for detecting abstract events from primitive events. It interacts with publishers and subscribers using the publish/subscribe primitives *(Subscribe(), Notify())* according to the following extension to the traditional publish/subscribe protocol: the AED Service publishes its interface, using an event service such as the CORBA Notification Service to all the subscribers and publishers of primitive events. Subscribers register their interest in subscribing to abstract event types of interest by subscribing to a dedicated *"AbstractEventSubscriptionListener"* (*AESListener()*) interface, at the AED Service. Each subscription carries the subscriber identification, an

AESL definition, and a filter[3].

$$\textit{AESListener}.\textbf{subscribe}(\textit{subscriberId}, \textit{AESL def}, \textit{filter})$$

The AED Service uses the AESListener to listen for subscriptions of the above type. For each received subscription, it checks in the *abstract event repository* whether an event type with the same name or AESL definition exists, and if so, adds the subscriber to the list of subscribers for this event type. If it doesn't already exist, the AED Service registers the new event type with the underlying event service, so that the abstract event is available for filtering. The AESL definition is made available to the abstract event repository along with the new abstract event type. Next, the AESL definition and filter are extracted in order to construct an abstract event detector that detects an abstract event of the requested type, as explained in Section 8.3.1. Using the AESL definition, the primitive events of interest are selected and the underlying notification service is used for subscribing to the primitive events which are then translated appropriately and forwarded as inputs to the abstract event detector.

Each time an abstract event is detected, *Notify()* is invoked in order to publish the abstract event. *Notify()* publishes both the abstract event and its AESL definition. This protects the service from malevolent event subscription in case of duplicate subscriptions to the same abstract event type with incorrect AESL definitions. The *AEUListener()* listens for *Unsubscribe()* requests and removes the client that corresponds to the unsubscribe event from the notification list for that abstract event type.

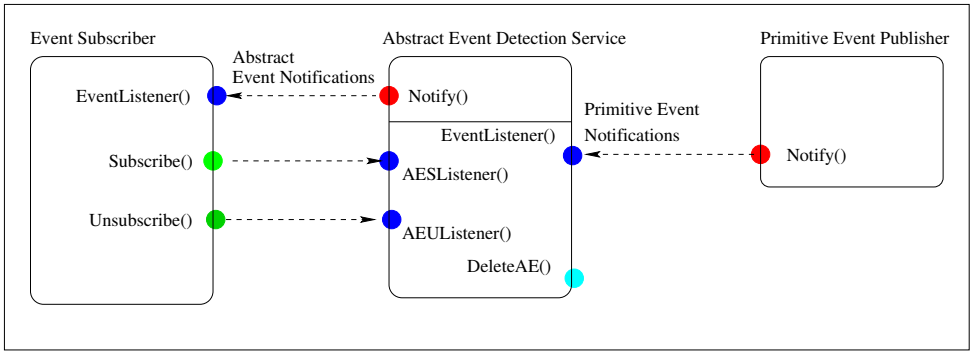


Figure 8.4. Abstract Event Detection (AED) Service.

The following call allows a subscriber to register interest in the events of type $H\_ClosestEmptyLocation$ (8.1). The filter of (8.2) is applied during subscription.

$$\textit{AESListener}.\textbf{subscribe}(\textit{appIOR}, \Phi_1, \Phi_2, \textit{type def})$$

It is assumed that the application is a CORBA object known to the system by its remote object reference (IOR). Φ_1 and Φ_2 correspond to (8.1) and (8.2) respectively.

8.4.1 Dynamic Retraction of Unused Abstract Event Types

When no subscribers are interested in a specific abstract event type, the detection of that abstract event type stops and any instances of the abstract predicate are garbage-collected from the knowledge base.

[3]This interface also contains as an argument an AESL type definition that is required by the underlying knowledge base implementation. Actually, the Subscribe() call is as follows: Subscribe(subscriberId, AESL def, filter, type def). Although AESL is a typed language, type definitions are ignored in this chapter, but are discussed in Chapter 12.

This functionality is implemented by the *DeleteAE()* interface, which deletes the abstract event detector that corresponds to the abstract event type of interest from the AED Service knowledge base. The detector is re-created the next time a subscriber exists for that event-type.

8.4.2 Satisfiability Checking

As AESL definitions are used as specifications for abstract event detection, it is desirable that they are checked for satisfiability, i.e., that there exists a state in the sensor-driven domain for which the binding definition is true. The opposite case, that there exists at least one state for which the binding definition is false, is also checked. Breaching of satisfiability can be caused by a user defining an abstract event that corresponds to a situation that is impossible, e.g., the request " Whenever the system administrator is at a different room from the health and security officer, notify me" is impossible if the system administrator is the same person as the health and security officer.

A prototype Satisfiability checking component was implemented using the theorem prover SPASS [101]. This service is discussed in Chapter 9.

8.4.3 Resource Discovery

A subscription log enables the AED Service to keep track of the availability or not of subscribers and publishers. If a publisher is unavailable, the AED Service can seek redundant sources for that context type. Furthermore, failure of the AED Service will cause the notification of subscribers and a global CORBA name service allows them to locate redundant services that provide the same functionality.

8.5 Distributed Abstract Event Detection

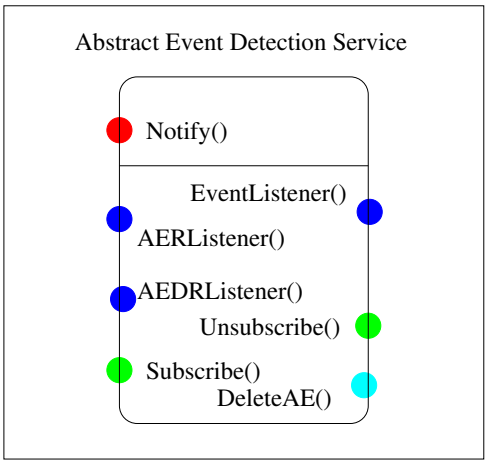


Figure 8.5. The interfaces of the distributed Abstract Event Detection Service component.

The AED Service was discussed in the previous section as a central entity, serving a number of event publishers and subscribers. In a large-scale distributed sensor-driven application, event publishers are situated in different physical domains which can be distributed in a wide-area setting. In such a case, a centralised AED Service will become a single point of failure. Instead, the AED Service is designed as a distributed component that is placed as close to primitive event publishers as possible. Note that this type of load balancing differs from the traditional concept, as the aim is not to define the optimal

server configuration for sharing the processing load, but rather the optimal places in the network where the event processing should take place in order to promote scalability.

Furthermore, AED Service distributed components can be combined hierarchically, sharing some of the processing. The interface of the distributed AED Service component is seen in Figure 8.5. The *Subscribe()* and *Unsubscribe()* interfaces are used in order for the component to subscribe to other AED Service components. The distribution of the AED Service component is made possible by the structure of the AESL definition statements. Because AESL definitions are structured as sets of rules, they are easily implementable hierarchically. For example, the AESL definition of Example 1 can be implemented with a network of AED Service distributed components as in Figure 8.6. AED_1 detects abstract events of type *H_EmptyLocation* (ae1), AED_2 detects abstract events of type *H_ClosestLocation* (ae2) and AED_3 events of type *H_ClosestEmptyLocation* (ae3).

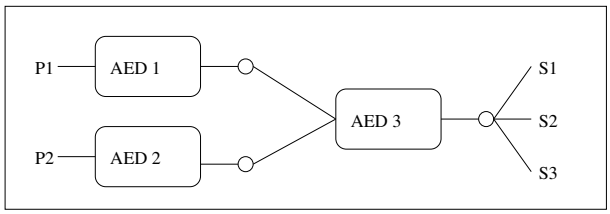


Figure 8.6. Hierarchical distributed AED Service architecture.

8.6 Analysis

This section describes a prototype scalability analysis of the extended publish/subscribe protocol, in terms of the number of events exchanged as well as the number of computational steps involved in the creation and dispatching of events. It is demonstrated that the overall event cost is significantly lower than that of a traditional publish/subscribe protocol. It is also demonstrated that the computational complexity cost is similar to that of traditional publish/subscribe. However, in the extended version, computation is not replicated unnecessarily and it can be easily accommodated on an appropriate server.

For simplicity reasons, a central architecture is assumed for the Abstract Event Detection service. The architecture consists of k domains, each containing on average n users and m rooms. It is assumed that *instances* of the concrete state predicates *L_AtomicLocation*, *H_UserInLocation*[4] and *H_Distance* are available directly as primitive events in each domain, by publishers p_1, p_2, p_3, respectively. These events are referred to as *AtomicLocation*, *UserInLocation* and *Distance* events, respectively.

8.6.1 Extended Publish/Subscribe

A subscriber application is interested in locating the closest empty room to each user in each domain (Example 1), and so it issues a subscription to the AED Service (see Section 8.4). The AED Service extracts the AESL definition from the subscription and uses it as a specification for generating the Rete network that corresponds to this abstract event. The Rete network is portrayed in Figure 8.2.

It is assumed that each user is moving at an event rate of 1 location event per second (which is a conservative average event rate produced by the Active BAT system). The AED Service subscribes to p_1 for *AtomicLocation* events that represent the topology of the domain in terms of the physical locations.

[4]Note that for simplicity reasons, it is assumed that location events that contain the region the user is in, e.g., *H_UserInLocation(Alan, Supervisor,FS15,Office)*, are available in the system.

AtomicLocation events will be published only once (m primitive events) as long as no new regions are added dynamically in the system[5] The AED Service subscribes to p_2 for *UserInLocation* events and receives, as a result, n notifications/sec that convey the changes to the location of each user so that the emptiness property can be determined. The application subscribes to p_3 for *Distance* events and thus receives $m \cdot n$ notifications/sec, which help determine the proximity property of each user to each room. The overall number of notifications from the publishers to the AED Service component is:

$$bandwidth_{p->AED} = k(n + mn)\ notifications/sec. \tag{8.4}$$

The AED Service maps the received notifications to instances of the corresponding predicates and propagates the changes inside the abstract event detector of Figure 8.2. Calculating the computational complexity of the Rete network of Figure 8.2, as in Chapter 7, the overall number of computational steps performed by this network are $O\left(m^2n^2\right)/sec$ [53]. A maximum of n tokens that describe the closest empty location for each of the n users are detected by the network and published by the AED Service as abstract event notifications.[6]

Event Bandwidth. The worst case estimate for the total event bandwidth required from the AED Service component to each subscriber amounts to:

$$\begin{aligned} bandwidth_{AED->s} &= O(kn)\ notifications/sec\ (overall) \\ &= O\left(n\right)\ notifications/sec\ (per\ domain) \end{aligned} \tag{8.5}$$

From Equations 8.4, 8.5, it can be inferred that as long as the AED Service is placed close to the primitive event sources, the event bandwidth from the sources to the AED Service will only contribute to the local network traffic. For the rest of the network, from the AED Service to the subscribers, the event bandwidth generated from the above scenario is $O(n)$.

$$bandwidth_{overall} = O\left(n\right)\ notifications/sec\ (per\ domain) \tag{8.6}$$

Computational Complexity. The maximum number of computational steps required at the server that accommodates the knowledge base at each domain is:

$$computational\ complexity = O(n^2m^2)\ \textit{computational steps/sec}\ (per\ domain)$$

Assuming that the number of users can grow rapidly and significantly exceeds the number of rooms, it is worth calculating the above complexity in terms of n only:

$$computational\ complexity = O(n^4)\ computational\ steps/sec\ (per\ domain) \tag{8.7}$$

8.6.2 Traditional Publish/Subscribe

Event bandwidth. In this case, it is the subscribing application itself that subscribes to sources p_1, p_2 and p_3, and therefore the overall notification rate that traverses the network from the sources to the

[5]For example, in the case of SPIRIT [41], a user can define a region in space which will cause his Bat to enter an energy-saving mode when placed in that region.

[6]In practice, this model uses two separate knowledge base layers, one for concrete state predicates and one for abstract ones, which together act as a two-layer state predicate cache (Chapter 6). This is ignored in this analysis. In a two-layer knowledge base, there are in average $n + 2$ events/second used for the communication between the two layers and $O(n)$ additional computational steps/second in the higher layer. These numbers do not affect the overall estimates.

subscriber is:

$$\begin{aligned} bandwidth_{overall} &= k(n+mn)\ notifications/sec\ (overall) \\ &= O\left(n+nm\right)\ notifications/sec\ (per\ domain) \end{aligned} \tag{8.8}$$

Assuming that there are more users on average than rooms, then:

$$bandwidth_{overall} = O\left(n^2\right)\ notifications/sec\ \ (per\ domain) \tag{8.9}$$

Computational Complexity. However, at the application side, additional computations are required in order to calculate the emptiness property and all subscribers need to perform these calculations per second. Assuming that calculations are done using the Rete algorithm, so that we can have a basis for comparison, then at least $0(m^2n^2)$ computations are replicated at each subscriber. Assuming that m is bounded by n, the above complexity becomes $O(n^4)$.

$$\begin{aligned} computational\ complexity &= O\left(m^2n^2\right)\ computational\ steps/sec\ \ (per\ subscriber) \\ &= O\left(n^4\right)\ computational\ steps/sec\ (per\ subscriber) \end{aligned} \tag{8.10}$$

8.6.3 Comparison

The overall event bandwidth in the case of the extended publish/subscribe (Equation 8.6) is significantly lower than that of the traditional one (Equation 8.9). Note that while the worst-case estimate of the computational cost in the traditional publish/subscribe (Equation 8.10) is similar to the extended one (Equation 8.7), the computation in the traditional case needs to be undertaken by each subscriber, whereas in the extended publish/subscribe computation takes place once, at the AED Service.

8.7 Related Work

Two other uses of *abstract events* have been proposed in the literature: Abstract events for distributed program execution [56, 57] and abstract events for representing processor states in a parallel-processing environment [24]. Both these uses are unrelated to the current effort.

A significant amount of effort has been expended already in the area of large-scale event-interaction based on the publish-subscribe paradigm. Most of this effort is focused on six design issues:

- Whether the event processing is performed in *one central unit* or in a set of *distributed servers* and in what *configuration* the servers should be connected.
- The message routing algorithm.
- The selection process that separates events of interest from all other available events.
- A processing strategy that determines the *optimal* places in the network that message data should be processed in order to *optimise message traffic.*
- Event composition.
- Middleware support.

As far as the first point is concerned, alternatives include a *centralised approach*, a *distributed approach* and a *distributed network of servers* approach [33]. In a centralised approach, (Oracle Advanced

Queueing and IBM MQSeries) a central entity called an *event service* is responsible for storing and forwarding messages, exchanged by *subscribers* and *publishers*. Applications based on such systems (banking, electronic commerce applications) have strong requirements in terms of reliability, data consistency and transactional support but do not need high data throughput. In a distributed approach such as the one used by TIBCO, producers communicate directly with consumers through a *store and forward mechanism* which is efficient for applications such as financial stock exchange. Lastly, Siena [15] and Hermes [79] use an intermediate approach, namely, a distributed network of *event brokers*, thus combining loose-coupling with persistent and reliable management of notifications.

As far as the second design strategy is concerned, different solutions deal with a trade-off between the complexity of the routing algorithm and the processing power of the distributed servers. Alternatives here include *broadcasting* [15] and *multicasting* [25, 93]. Scribe [93] uses an efficient peer-to-peer routing method [92] for the design of a scalable notification-dissemination platform using *rendezvous* nodes.

Thirdly, because subscribers are usually interested in particular events or event patterns, and not in all events that are being published, *filtering* can be used in order to identify the notifications of interest for a given subscriber. As a consequence, filtering leads to the reduction of the overall event bandwidth. This is implemented by the OMG group in their Event Notification Specification [61].

Fourthly, several filtering techniques can be used in order to further reduce the number of delivered events. Siena proposes a novel *processing strategy* which reduces the number of exchanged notifications, by using a mechanism that observes *similarities* between subscription filters and forwards through the event architecture only those subscriptions that are not included in a previous, more general subscription pattern. Hermes implements a similar mechanism. Furthermore, Siena abides by the principle that filtering should be applied as close as possible to the sources of notifications, whereas a notification should be routed in one copy as far as possible and should be replicated as close as possible to the parties that are interested in it. This is similar in concept to the combining filters in SCAFOS, used to avoid replication of computation (see Chapters 8 and 12).

Significant work has also been done in the area of *event composition* [6, 7, 63, 102] , i.e., the combination of primitive events into composite events by applying a set of *composition operators*. However, event composition does not offer sufficient expressiveness for the requirements of many applications, such as the ones in Sentient Computing, nor does it hide the complexity of the event architecture from the application layer. Using event composition, in order for an application to register interest in an abstract situation, it would have to register for all *compositions* that lead to the abstract state of interest. For example, consider an application that is interested in receiving notification about the *closest, empty meeting room to a mobile user*. Using event composition, the application would have to register for all combinations of events that would lead to the above situation, e.g., it would have to include events that lead to all the possible ways that people have moved in order for a meeting room to be empty, e.g., even for people who have previously moved into the meeting room and out of it again.

Hermes appears to take event interaction a step further and looks at the publish/subscribe protocol as the basis for an *event-based middleware architecture*, providing support similar to traditional middleware systems (CORBA, Java RMI) in order to address issues such as *fault-tolerance, type-checking of invocations* and *reliability*. Furthermore, Hermes aims to improve the efficiency of the delivery of event notifications by using peer-to-peer routing techniques for creating overlay broker networks [80]. *Role-based access control* for publish/subscribe middleware architectures is the objective of [10] which discusses an integration of the Hermes middleware and the OASIS [8, 43] framework.

A generic middleware component that undertakes the process of correlating and aggregating events is presented in [46]. The contribution of this work is that it provides a fast, computationally efficient way for the subscriber to determine when a situation of interest is satisfied, which is at the same time generic and independent of the type of situation of interest.

The seminal notion of abstract events in sensor-driven systems appears to have originated in [7] and it is mentioned again later in [6]. However, this early intuition is case-specific, static and not directly

applicable to loosely coupled systems. In both publications [7, 6], point to point communication between the *event source* and the *event client* via an *event mediator* is assumed, forcing close coupling between the event source and the event client. These efforts do not discuss the semantics of programmable support for the dynamic creation of abstract events in an asynchronous manner.

In conclusion, existing efforts are not directly applicable to sensor-driven systems because they lack the ability to deal with very large-scale event interaction, dynamic extensibility and a natural interface that hides the underlying event implementation.

8.8 Conclusions

This chapter extends the work presented in Chapter 5, which advocates state-based modelling for context-awareness in sensor-driven systems by demonstrating that subscribing to changes in system state is more appropriate for sensor-driven systems than subscribing to event sequences that lead to such state. It proposes an extension to the publish/subscribe protocol that allows transparency in subscribing to distributed state. Chapter 5 demonstrated that current event models have limitations in this respect, which are due to the incomplete mapping between the subscription language (FOL) and the elements of the implementation domain, as well as the insufficiency of finite automata to deal with negation and quantification. In order to address these limitations, a state model was presented in Chapter 5. The proposed model is implemented using a deductive knowledge base, which deduces the predicates that complete the above-mentioned mapping. In this chapter, the concept of abstract events as changes of abstract state is introduced and a higher-order service is described. This service takes as an argument an abstract event specification and, in return, publishes an interface to a further service, an abstract event detector, which notifies transitions between true and false values of the specification. In this way, scalability is promoted, and the computation is placed closer to the publishers, avoiding unnecessary replication.

Although the scope of this work has been sensor-driven systems, the enhanced publish/subscribe protocol presented here can be used with all event systems where abstract events make sense.

Chapter 9

Model Checking for Sentient Computing: An Axiomatic Approach

Previous chapters introduced a state based model and a scalable abstract reasoning scheme for context-aware knowledge in sensor-driven systems. This chapter investigates the *correctness* of the *model* that represents the current state of the dynamically changing world as well as its *semantic compatibility* and *interoperability* with context-aware applications. This model can be seen as a *concrete* interpretation of the physical environment and conceptually stands between the physical world and the *abstract* view of the applications. A number of factors such as the non-homogeneity of physical space and the precision of the sensor technology may introduce errors and inconsistencies between the physical world and the model. On the other hand, the abstract view of the application domain needs to be correct and compatible with the concrete model, especially in the case of distributed environments where applications need to interact *seamlessly* with several different concrete model components.

This chapter proposes a system, similar to a model-checker, that checks the *satisfiability* of the application requirements against the physical environment model, as well as the consistency of the model with the physical environment, thus promoting distribution. The implementation of the proposed system is based on a theorem prover.

9.1 Introduction

Chapters 5, 6 and 7 discuss various aspects of a model for sensor-driven systems where abstract knowledge is deduced from concrete knowledge according to AESL definitions. However, this dual *abstract mapping*, i.e., from the physical environment to the concrete model and from the concrete model to the application layer, can be affected by a number of factors in terms of *correctness, completeness and consistency* of both the model and the application specifications. The most important of these factors are summarised below:

- *Non-homogeneous space and the natural laws of physics*. The application specifications may contain logical fallacies that are caused by ignoring the physical constraints that are introduced by the spatial topology, i.e., a person cannot be in more than one location simultaneously, and he cannot move through walls.

- *Semantic sufficiency of the model*. The application is not aware of the correctness or the granularity of the model of the physical world that depends directly on the capabilities of the underlying location system. For example, a model that is updated by Active Badge [106] location sightings only knows about rooms. Such a model cannot reason with any application requirements that involve positions or regions smaller than rooms. For example, such a model cannot determine to

which PC in the room the user is closer, and therefore such an application requirement would be *unsatisfiable* by the specific model. However, a model that depends on the Active BAT [41] for location information knows the exact position of a user and can deduce much more abstract state in order to satisfy the application requirements.

- *Correctness of abstract application specification*. The application specification may be incorrect, in which case it will not be satisfiable by the model even if the model is semantically adequate. Errors can occur from violating logical constraints, such as the ones that are derived from the functional operation of location predicates. For example, the situation in which the C expert is typing at his keyboard while at the same time the systems administrator is having a coffee is impossible if the system administrator and the C expert are the same person. Feedback should be returned to the application in such a case.

Model checking [18] was proposed in 1981 by Dr. Edmund Clarck as a method for formally verifying finite-state concurrent systems, where specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not. We propose a verification technique similar in concept to model checking that checks abstract knowledge definitions (AESL definitions) for correctness and semantic compatibility with the models of the sensor-driven domains. Correctness is established against a set of spatio-temporal constraints and against a set of logical constraints that make sure that logical fallacies are excluded. The Sentient model is also checked for correctness by checking each sensor update against a set of spatio-temporal specifications. Whenever a specification is found unsatisfiable, appropriate feedback can be given to the application that can be used for the selection of a more appropriate model, or an adaptation in the application's behaviour.

9.2 Factors that Affect Modelling

The main factors that make the concrete model incomplete or inconsistent with the physical environment is the combination of the behaviour of the sensor technology that instruments physical space and the non-homogeneity of physical space itself. The main sensor technology used for physical space modelling is *location* technology. *Location technologies* vary in precision, and an imprecise or an incorrect reading will introduce an error in the concrete model. Because space can differ so much from one cm^3 to another, such an error can lead to an *illegal* model state. For example, a user can be seen floating in the air, or walking inside a wall.

Similarly, the application abstract view can be incomplete or inconsistent compared to the concrete model. Inconsistencies here can be introduced via incorrect application specifications e.g., an application specification may describe an abstract state where the application is interested in paging the first available C++ expert, while the system administrator is unavailable. This particular abstract state cannot be reached, as long as the system administrator is the only C++ expert.

Another constraint derives from the situation in which an application interacts with more than one concrete model at different levels of *knowledge* precision. For example, the above application specification would be *unsatisfiable* for a model that only knows which room a user is in and therefore cannot determine whether one is using a PC or not. The proposed system prevents the above problems by checking the concrete model against a set of specifications that prevent the modelling of illegal states.

9.3 Specifications

Specifications are described in terms of FOL axioms, and can be classified into three sets: *Spatial and logical* specs, *Model* specs and *Application* specs. *Spatial and logical specs* are used to check the consistency of the model against the physical environment. They represent physical constraints that apply

to all Sentient environments (see Section 9.2). *Model* specs are used to determine whether the abstract specifications are compatible with the concrete model or whether the model is semantically sufficient for the application specification. *Application* specs [1] are provided by the applications in the form of axioms and aim to bind concrete state predicates to a high-level, abstract state predicate, referred to in this paper as the *goal theorem*. Goal theorems are checked for satisfiability against application specs, model specs and spatial and logical specs. Moreover, each sensor update is checked for satisfiability against spatial and logical specs.

Overall Specification

The satisfiability of the overall specification is modelled by the predicate $SpecSuccess$, which is defined as equivalent to the conjunction of the predicates that signify the satisfiability of the application model and spatial specifications, respectively:

$$AppSuccess \wedge \neg ModelFailure \wedge SpatialSuccess \quad \Rightarrow \quad SpecSuccess$$

9.3.1 First-Order Logic Description of a Sentient Model

Consider a Sentient environment with j users (uid) that share m roles ($role$) and move around n locations (rid) that can be characterised by u attributes ($rattr$). Using the model definitions of Chapter 5, facts are of type $L\_AtomicLocation(rid, rattr, polygon)$, $H\_UserInLocation(uid, role, rid, rattr)$, or $H\_Distance(v, uid, role, rid, rattr)$.

9.3.2 Spatial and Logical Specifications

Spatial and Logical Specifications aim to represent constraints that derive from the characteristics of physical space (see Section 9.2). Both application-provided theorems and new facts produced by the sensor infrastructure are checked against this set of specifications. The following axioms represent such specifications:

Axiom 1. *Each user can be in only one position at a time.*

$$\begin{aligned}&(x_1 = x_2) \wedge (y_1 = y_2) \wedge (z_1 = z_2)\\ &\vee \neg L\_UserAtPosition(uid, role_1, x_1, y_1, z_1)\\ &\vee \neg L\_UserAtPosition(uid, role_2, x_2, y_2, z_2)\end{aligned}$$

The above axiom prevents the application from making a specification that implicitly requires the same user to be in two different positions at the same time, usually under a different role, e.g., a user who is both a system administrator and a C++ expert. We can also say that $L\_UserAtPosition$ is a *function* in the domain of the concrete model. A direct consequence of the above axiom is the following:

Axiom 2. *If an object is contained in more than one region, these are nested.*

$$\begin{aligned}&(L\_NestedLocation(rid_2, site\text{-}list_2) \wedge InList(rid_1, site\text{-}list_2))\\ &\vee (L\_NestedLocation(rid_1, site\text{-}list_1) \wedge InList(rid_2, site\text{-}list_1))\\ &\vee \neg H\_UserInLocation(uid, role_1, rid_1, rattr_1)\\ &\vee \neg H\_UserInLocation(uid, role_2, rid_2, rattr_2)\end{aligned}$$

[1] Application specs are equivalent to AESL definitions.

The above axioms denotes that if a user is known by the system to be inside two different regions then the regions are nested [2].

Axiom 3. *A user cannot be located inside a compact surface such as a wall.*

$$IsOpaque(x_1, y_1, z_1) \Rightarrow \nexists L\_UserAtPosition(uid, role, x_1, y_1, z_1)$$

This axiom can be used to discover an erroneous location system sighting. The predicate $SpatialSuccess$ is used in order to denote that all axioms in this category are satisfied.

9.3.3 Model Specifications

Model specifications are used in order to determine whether the concrete model is precise enough for the application specification. Model specifications are provided both by the application and the sensor-driven model side. The application provides an atomic formula:

$$RequiredPrecision(x, y)$$

The pair x, y is a confidence level that characterises the precision of the location modelling. The term x takes values from the set $\{region, coordinates\}$ and denotes the type of the location technology, and $y \in [1 \cdots 10]$ represents the accuracy of the coordinate system[3]. The higher the accuracy, the higher y. The concrete model provides the atomic formula:

$$ProvidedPrecision(x, y)$$

The pair x, y is also a confidence level [4] for location modelling precision. Similar predicates can be invented for other types of information.

Axiom 4. *The knowledge precision required by the application should be no greater than the knowledge precision offered by the underlying model.*

$$\begin{aligned}
& RequiredPrecision(region, 0) \wedge ProvidedPrecision(region, 0) \Rightarrow \neg ModelFailure(x, y) \\
& RequiredPrecision(region, 0) \wedge ProvidedPrecision(coordinates, y) \Rightarrow \neg ModelFailure(x, y) \\
& RequiredPrecision(coordinates, x) \wedge ProvidedPrecision(coordinates, y) \\
& \wedge > (x, y) \\
\Rightarrow\ & ModelFailure(x, y) \\
& RequiredPrecision(coordinates, x) \wedge ProvidedPrecision(coordinates, y) \\
& \wedge > (y, x) \Rightarrow \neg ModelFailure(x, y)
\end{aligned} \tag{9.1}$$

Initially the predicate *ModelFailure* is set to true. Each axiom which is satisfied sets it to false. If *ModelFailure* equals false, the model specifications are satisfied.

[2] It is assumed that no regions are overlapping.

[3] If x="region"then y=0 by default.

[4] Here, it is assumed either coordinate or containment granularity for the location system, and a confidence level for the precision of the coordinate system in 95% of the measurements.

9.3.4 Abstract Knowledge Definitions (AESL Definitions)

Assume an application that needs to determine the closest empty meeting room to the CTO of a company so that it can initiate a tele-conference. It is, therefore, interested in knowing when the predicate $ClosestEmptyLocation(uid, CTO, rid, Meeting\ Room)$ is true, as well as the value of rid. Because such a predicate does not exist in the system, the application needs to bind this to a set of facts that the system knows about in a logical way.

The *AESL definitions* can be seen below (for reasons of simplicity we assume that the predicate $ClosestLocation$ is calculated by the system using the predicate $Distance$).

$$\begin{aligned}
& RequiredPrecision(region, 0) \\
& \nexists uid\ H_UserInLocation(uid, role, rid, rattr) \Rightarrow H_EmptyLocation(rid, rattr) \\
& H_Distance(v_1, uid, role, rid_2, rattr_2) > H_Distance(v_2, uid, role, rid_1, rattr_1) \\
\Rightarrow\ & H_ClosestLocation(uid, role, rid_1, rattr_1) \\
& H_ClosestLocation(uid, rid, role, rattr) \wedge H_EmptyLocation(rid, rattr) \\
\Rightarrow\ & H_ClosestEmptyLocation(uid, rid, role, rattr) \qquad (9.2)
\end{aligned}$$

The goal theorem is the theorem that needs to be checked for satisfiability. Satisfiability of the goal theorem implies the satisfiability of the application specification, i.e., the predicate *AppSuccess*, which is initially true, remains true. In this case,

$$\exists rid(H\_ClosestEmptyLocation(uid, CTO, rid, Meeting\ Room)) \Rightarrow AppSuccess \qquad (9.3)$$

9.4 Proof by Resolution and Satisfiability

The contribution of this work is based on using the concept of *satisfiability* as the foundation for evaluating the conformance of application specifications and sensor updates to the Sentient model.

A *satisfiability* problem in conjunctive normal form (CNF) consists of the conjunction of a number of clauses where a clause is a disjunction of a number of variables or their negations. Given a set of clauses $C_1, C_2, \cdots C_m$ on the variables $x_1, x_2, \cdots, x_n$, the satisfiability problem is to determine if the formula

$$C_1 \wedge C_2 \wedge \cdots \wedge C_m$$

is satisfiable, that is if there is an assignment of values to the variables so that the above formula evaluates to true. Clearly, this requires that each C_j evaluates to true.

Automatic theorem provers aim to find a *proof* for a theorem given a set of axioms that are known to be true. A common method for finding a proof is by *resolution*. According to proof by resolution, the set of axioms and the goal theorem are transformed to conjunctive normal form (CNF), and resolution is applied to the resulting set of clauses. Existence of proof is equivalent to the satisfiability of the set of clauses.

9.4.1 The Theorem Prover SPASS

SPASS [107] is a first-order logic theorem prover with support for equality, and it was used in a prototype implementation of a satisfiability service whose API is shown in Figure 9.1. SPASS also features a Web interface (Web SPASS) [101], as well as integrated support for transforming FOL formulas into a small number of conjunctive normal form (CNF) clauses [75] before testing for unsatisfiability.

9.4.2 Example

Let us assume that the above application specification needs to be tested against a model which knows of locations in terms of coordinates, with accuracy of 3 cm, in 95% of the cases, such as the Active BAT. This can be encoded with the predicate $ProvidedPrecision(coordinates, 9)$. When tested with SPASS, the goal theorem is found to be correct.

```
SPASS V 2.0
SPASS beiseite: Proof found.
Problem: /tmp/webspass-webform_2003-09-09_00:40:50_29751.txt
SPASS derived 2 clauses, backtracked 0 clauses and kept 57 clauses.
SPASS allocated 528 KBytes.
SPASS spent     0:00:00.33 on the problem.
                0:00:00.04 for the input.
                0:00:00.09 for the FLOTTER CNF translation.
                0:00:00.02 for inferences.
                0:00:00.00 for the backtracking.
                0:00:00.04 for the reduction.
```

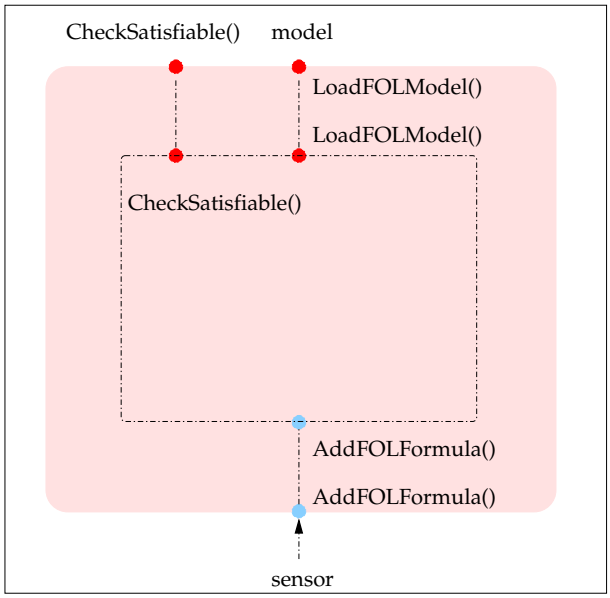


Figure 9.1. The Satisfiability Service and its API.

The *LoadFOLModel()* interface is used for loading SAL facts that denote the specific implementation of each domain, in terms of the topology, the number of users and the specific requirements of the instrumentation technology. The *CheckSatisfiable()* interface checks the satisfiability of a logical formula that represents an application definition, as described in this chapter. Such application definitions correspond to AESL definitions discussed in Chapter 12. Lastly, the *AddFOLFormula* interface asserts a SAL fact derived from a sensor and checks this for consistency with the FOL model.

9.5 Conclusions and Further Work

A service that checks Sentient Computing specifications for correctness and semantic compatibility with Sentient models was presented in this chapter. Three types of correctness specifications have been dis-

cussed. The service is based on the theorem prover SPASS, and programming logic has been developed that ensures that all specifications are satisfied for the overall application specification to hold.

Further work includes:

- Evaluating the satisfiability of the temporal properties of the AESL definitions, in addition to the logical ones. This requires support in the theorem prover for mathematical evaluation such as equality, inequality, etc.
- The performance of the Satisfiability Service, when used for evaluating real-time sensor updates, remains to be tested.

Chapter 10

SCAFOS: A Framework for the Development of Context-Aware Applications.

This chapter describes an infrastructure that facilitates the creation and deployment of applications that satisfy the requirements for context-awareness in sensor-driven systems, as stated in Chapter 5. The user provides a high-level specification of the application to be developed using SCALA (Chapter 12). SCAFOS undertakes the burden of the end-to-end implementation, from the user to the sensor layer. During the end-to-end processing, SCAFOS checks the correctness of user requirements, decides whether the specification is feasible given the underlying components, and, if it is, it requests the creation of the necessary events and subscribes to them. It then monitors the distributed components of interest, and generates the abstract state that the application is interested in from aggregated, concrete and abstract state that is available in the model. Once the specification is met, the desired system response is triggered. This chapter discusses the work done in implementing SCAFOS and the insights gained from the implementation.

10.1 SCAFOS

The SCAFOS framework is a prototype implementation of the model proposed in chapters 5 to 7. For the most part, the implementation is based on Java, as this is a widely-used language that can be compiled to different platforms. The detailed SCAFOS architecture is portrayed in Figure 10.1. This architecture consists of a set of components that provide the functionality described in this thesis. These are outlined here:

- The ECA Service allows the development of context-aware applications. Its design is inspired by initial work presented in [59] but it has been extended and thinned down in order to be integrated with SCAFOS. The functionality of the ECA service is discussed in more detail in Chapter 10.

- The Satisfiability Service implements the service described in Chapter 8. It checks the correctness of the specification in terms of abstract knowledge and addresses any conflicts between the specification and the implementation.

- The Abstract Event Detection Service component implements the service described in Chapter 7, i.e., a higher-order service that creates publishers for abstract events according to the specifications. The AED Service can be distributed to more than one component, as discussed in Chapter 7.

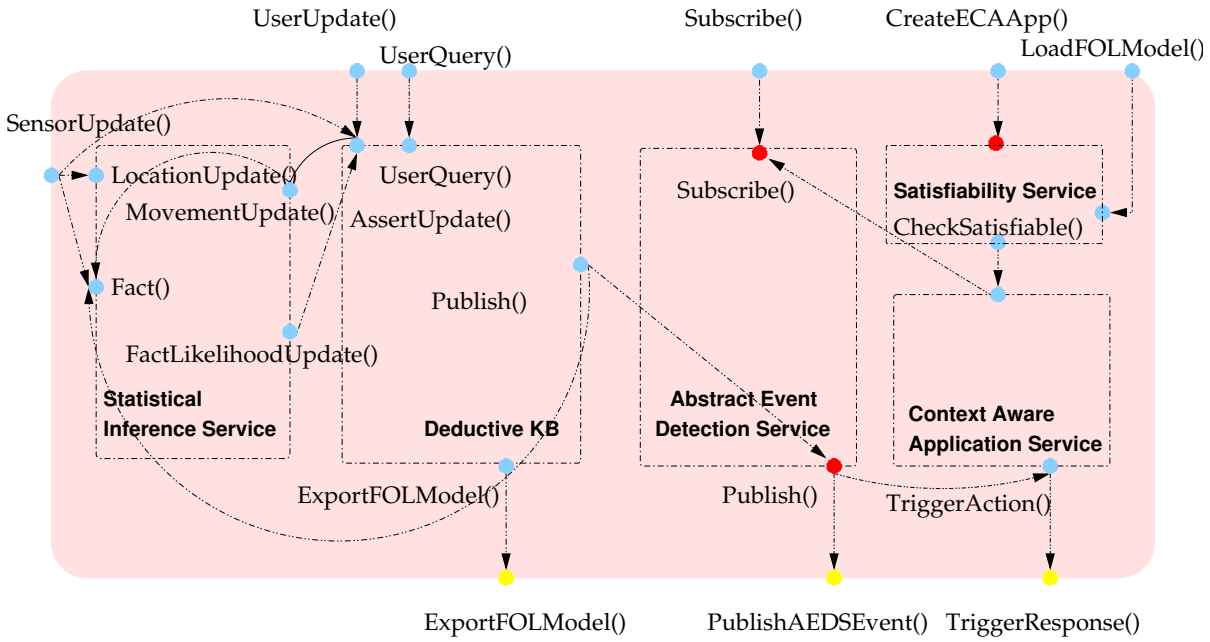


Figure 10.1. The SCAFOS conceptual framework.

- The Deductive Knowledge Base (KB) component implements the model for abstract and scalable reasoning of Chapter 6. More than one Deductive Knowledge Base component may operate in the infrastructure for reasons of distribution (see Section 10.1.1).

- The Statistical Inference Service supports movement recognition and likelihood estimation, as discussed in Chapters 3 and 4.

10.1.1 Distribution and Transparency

Transparency in distributed communication is ensured by using CORBA technology in order to implement the components of the architecture. *Resource discovery* of the SCAFOS services is enabled by using a global CORBA name service. The AED Service maintains a subscription log that allows it to check for the availability of publishers and subscribers via the name service. Subscribers can also check the availability of the AED Service and seek redundant resources.

The Deductive KB component in Figure 10.1 is implemented as a CORBA object, which makes it accessible transparently via multiple remote clients, such as the *Abstract Event Detection Service*, the *Statistical Inference Service*, the *Satisfiability Service* and the user interfaces *Subscribe()*, *UserQuery()* and *UserUpdate()*. Communication between the Deductive Knowledge Base component and the rest of the components is asynchronous using the *CORBA notification service*. However, inside the Deductive KB component, communication is achieved asynchronously by passing strings as remote invocations. Figure 10.2 illustrates the use of the interfaces *Subscribe()*, *UserQuery()* and *UserUpdate()*. These interfaces are *exported* to a central SCAFOS interface, which, whenever it is invoked, triggers the invocation of the appropriate interfaces in each component. For example, when the SCAFOS interface *UserQuery()* is invoked by the query *locate an empty room which is closest to the CEO*, all the respective *UserQuery()* interfaces in all the relevant Deductive KB components will be invoked.

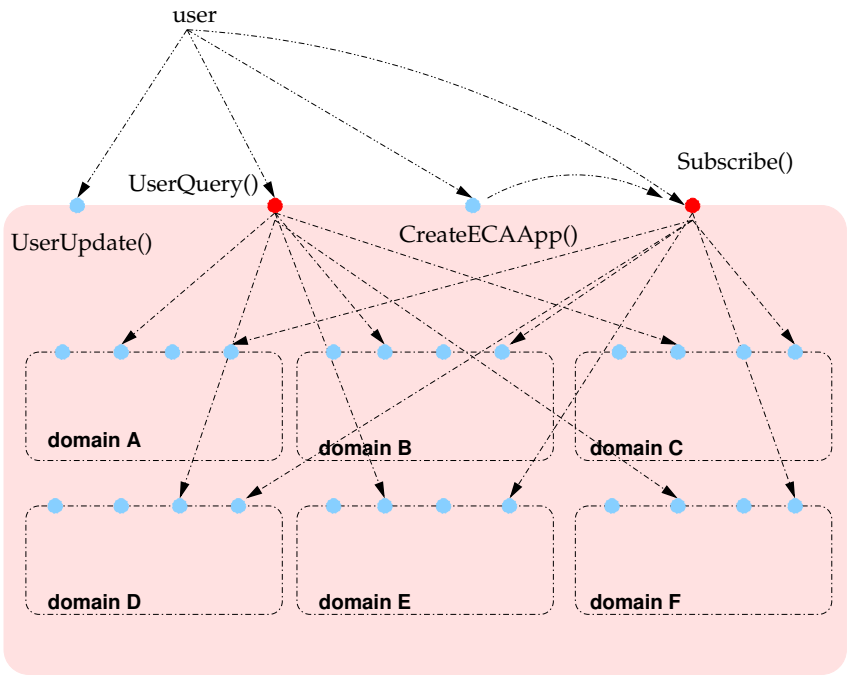


Figure 10.2. Deductive KB distribution

10.1.2 Concurrency

The Deductive KB component has been implemented with concurrency in mind. The Interfaces *UserQuery()*, *CreateAESLDef()* and *UserUpdate()* use *Read* and *Write locks* to ensure that integrity similar to the *Multiple Readers - Single Writer* problem is achieved. SCAFOS runs independently of the lifecycle of the applications that are created in the framework. Context acquisition, application creation, subscription and queries are constantly available. This is made possible because the Deductive KB component is threaded. A new listener thread is spawned that listens for *CreateAESLDef()*, *UserUpdate()* and *UserQuery()* calls, every time such a call is processed by the Deductive Knowledge Base component. The same applies to the *Subscribe()* interface of the AED Service component and the *CreateECAApp()* interface of the ECA Service component.

10.1.3 Maximum Integrability

The implementation of SCAFOS is based on Java which is a highly integrable platform. The knowledge base component is written in Jess [51], which is a Java shell for the expert system CLIPS [19]. Communication between Jess and Java is achieved by passing string-based events. Data is extracted from the sensor readings and an appropriate string is constructed. Appropriate listeners and event publishers can be written for any sensor technology. CORBA events produced by the Active BAT and Active Badge sensor readings have been used in this scheme.

10.2 The Deductive KB component

The prototype implementation of SCAFOS is described in more detail here. It consists of the following logical components. (Figure 10.3):

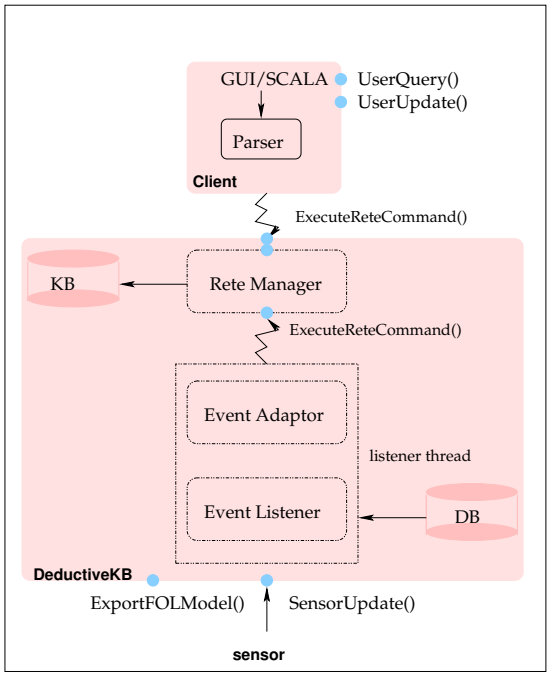


Figure 10.3. Software components architecture.

A Knowledge Base Component. A knowledge base is implemented in Jess [51].

A Database Component. The database component holds the predicate definitions of the model of Chapter 5, as well as a set of initial facts that describe the entities in the local domain such as its topology (SAL predicates *L_AtomicLocation*, *L_NestedLocation*, *L_InRegion*). As static knowledge held in the database is separated from the dynamic knowledge that is also held in the knowledge base, a single database component can be used in order to load the model and initial configuration into more than one knowledge base component, thus achieving load-balance. If persistence is required, the database can be used to store snapshots of the dynamic knowledge stored in the knowledge base. In the prototype implementation, the database is implemented in *mySQL* [70], which is an open source database technology. An object-wrapper, implemented in *Castor* [16], operates above this, providing an object interface to the underlying database.

A Rete Manager Object. The *Rete Manager* is implemented as a CORBA object. This object manages all accesses to the knowledge base by means of remote invocations. The *Rete Manager* exports an IDL interface that consists of two methods: *ExecuteReteCommand(string command)* and *ExecuteReadReteCommand(string command)*. The *ExecuteReadReteCommand(string command)* method contains a Read lock so that multiple threads can access the same knowledge base concurrently. The *ExecuteReteCommand(string command)* contains a write lock so that multiple threads can access the same resource exclusively. Both methods take a string which represents the command as an argument and pass it to Jess where the command is asserted in the knowledge base. Jess raises a number of exceptions that are captured by the Rete Manager if the assertion is unsuccessful. Referring to Figure 10.1, it is worth noting that the interfaces *UserQuery()*, *Subscribe()* and *UserUpdate()* are only implemented through the IDL interface *ExecuteReteCommand()*.

Inside the Deductive Knowledge Base component, multiple knowledge base components and, consequently, Rete Manager objects (Figure 10.3) can be created, initialised and used. This enables the

layering of the knowledge base (see Chapter 6) and allows for efficient load-balancing algorithms to be used.

An Event Adaptor Object. An *Event Adaptor* object translates sensor-derived events (from SPIRIT and QoSDREAM) to Jess facts, and asserts these in the knowledge base.

A Listener Thread. The listener thread consists of the entities *EventListener* and *EventAdaptor*. The *Event Listener* listens for location events produced by the Active BAT and the Active Badge (using QoSDREAM). It has filtering capabilities on event type and event attributes. Once an event is received, it extracts the object attributes from the event. It then passes the object and the location data to the *Event Adaptor*. The *Event Adaptor* is a CORBA client object that performs the translation between the received event and a string and performs a remote invocation on the *Rete Manager*. It reports any exceptions raised. Two *Event Listener* components have been built that listen for Active BAT and Active Badge events, respectively. Several such threads may operate concurrently.

```
class EventListener extends EventClient
{
        public EventListener(EventDescriptor eventDes) throws Exception
{
            super(eventDes,null);
        }

        public void push_structured_event(StructuredEvent event)
        {
System.out.println("Got an event!");
        Vtimer++;
        System.out.println("Vtimer= " + Vtimer);
        Person next=null;
           try {
               org.omg.CosNotification.Property[] values=event.filterable_data;

                int locatableId = values[0].value.extract_long();
                int regionId = values[1].value.extract_long();
                int oldOverlap = values[2].value.extract_long();
                int newOverlap = values[3].value.extract_long();
}
```

A User Interface. A prototype user interface has been built which allows users to assert rules (AESL definitions) into the knowledge base, through the Rete Manager object. Two different approaches have been implemented. In the first approach, the user asserts a string in SCALA that is parsed into a set of CLIPS rules, which are asserted through the *ExecuteReteCommand()* interface into the knowledge base component. In the second approach, an object-oriented representation is used for the model of Chapter 5. In this scheme, each predicate is modelled by a Java Bean [50] class, and an instance of this class is generated for each predicate instance. Because Beans can be easily portrayed graphically, this approach has the advantage that a GUI can easily be constructed directly from the model. The implementation of the UserLocationBean is shown below for illustration.

```
public class UserLocationBean extends JPanel implements ActionListener,
   Serializable

     public String b_name="UserLocationBean";
     public int bid=0;
     private String type = "Movement";
     private int u_id= 0
     private String name="" ;
     private String surname="";
     private String username="";
     private Region office= null;
     private String phone="";
     private String email="";
```

```
     private String imageFilePath= "/home/ek236/photos/";
     private String imageFileName;
     private String ulocation= " " ;

     private String role=" ";
     private float timestamp=0 ;
     protected ImageIcon image;
     public UserLocationBean(){
     }
     public UserLocationBean(String icon) {
     this.setImageFileName(icon);
     image = new ImageIcon(icon);
     }
     public String getName() {return name; }
     public String getSurname() {return surname;}
     public String getUlocation() {return ulocation;}
     public void setName(String s) { name = s; }
     public void setSurname(String s) {surname = s;}
     public void setRole(String r){role=r;}
     public void setUlocation(float z){uz=z;}
}}
```

A Messaging Client. A client that delivers messages via an email server and SMS messages via an email-sms gateway has been developed using the *Userfunction* interface of Jess. The client is triggered from the knowledge base as a result of the firing of a rule. The Java interface *jess.Userfunction* represents a single function in the Jess language. The developer can add new functions to the Jess language simply by writing a class that implements the *jess.Userfunction* interface, creating a single instance of this class and installing it into a jess.Rete object using *Rete.addUserfunction()*. The *Userfunction* classes can maintain state; therefore a *Userfunction* can cache results across invocations, maintain complex data structures, or keep references to external Java objects for callbacks. A single *Userfunction* can be a gateway into a complex Java subsystem. For the implementation of the interruptibility manager using JESS, the *jess.Userfunction* interface has been used to integrate JESS with the rest of the Java code responsible for event reception (Event manager) and action execution (Notification Execution). The following example illustrates how the *jess.Userfunction* interface has been implemented to integrate Jess with the notification server.

```
import java.io.*;
import jess.*;
import  java.net.*;
import  java.io.*;
import  java.util.*;
import Smtp;
import Mail;

public class SendEmail implements Userfunction
{
  //The name method returns the name by which the function will appear in Jess.

   public String getName()
   {
return "sendmail" ;
  }

   public Value call (ValueVector vv, Context context) throws JessException
{
try{
    Smtp smtp = new Smtp("wrench.eng.cam.ac.uk");
            String addr = vv.get(1).stringValue(context);
    String sub = vv.get(2).stringValue(context);
            Mail  email= new  Mail("ek236","Prof. Hopper has entered the LCE");
            email.start();
            } catch (Exception e){
              System.err.println(e.getMessage());
```

```
            }
    return new Value ("mail sent!", RU.STRING);
        }
}
```

The *call()* method is the core of the Userfunction. When *call()* is invoked, the first argument will be a ValueVector representation of the Jess code that invoked the function. For example, when the following Jess function calls are made,

```
Jess> (load-function SendEmail)
Jess> (sendmail ek236 ‘‘Prof. Hopper has entered the LCE.’’)
```

an email is sent to user ek236 with the subject: “Prof. Hopper has entered the LCE.”

10.3 Conclusions

The SCAFOS framework, which implements the model of Chapters 5 to 7, has been described. SCAFOS has the following properties:

- It facilitates the creation and deployment of context-aware applications in sensor-driven systems. Context-aware applications that operate in SCAFOS are automatically integratable with existing context-aware services, and they satisfy all the requirements of Chapter 5.
- It is threaded, so it guarantees constant context acquisition and constant availability of the *CreateECAApp(), UserQuery(), Subscribe(), UserUpdate()* interfaces.
- It promotes scalability, portability, extensibility and an easily constructible user interface. It is easily integrable with heterogeneous software technologies.

Chapter 11

Applications

This chapter presents an evaluation of SCAFOS by discussing the implementation of four example applications.

11.1 Experimental Setup

In order to evaluate SCAFOS, the following experiment was carried out: the movements of 20 users in the old LCE were monitored using the Active BAT location system for a period of 40 hours and 48 minutes, starting at 21:20 on day one and finishing approximately at 14:08 on day three of the experiment. During that time, a total of 30,612 events were received. These events were published directly by SPIRIT, and therefore they correspond to the *L_UserInLocation* predicate rather than the *L_UserAtPosition* predicate, i.e., they contain the user's location in terms of a room, rather than the user's position in terms of coordinates. For example, a "sighting" of the user *Dave Scott* in *Room 10* at *13:45* is modelled by the event

L_UserInLocation(Dave Scott, Room 10, 13:45).

The *L_InRegion* predicate (Chapter 6) is implemented internally in SPIRIT. The *UserInLocation* predicate is used both with the prefix *L_* and *H_* depending on whether it is seen as a high-level predicate deduced from the *L_UserAtPosition* predicate or a low-level predicate received directly from SPIRIT as is the case in this experiment.

A set of libraries were loaded in SCAFOS' deductive knowledge base component in order to generate the system's desired behaviour. Two of these libraries were used for modelling the old LCE environment (a total of 29 Jess rules). The first library contained rules to define the old LCE's spatial topology. These rules generate instances of the SAL predicates *L_AtomicLocation* and *L_NestedLocation*. These define the available rooms, floors and laboratories (Old-LCE, Fallside, GRO), etc., and the attributes associated with them. For example, *Room 7* is associated with the attributes *Andy's Office, Office, Supervision Area*. *Room 7* is also contained within *Floor 5* which is in turn contained within *Old-LCE*, *Engineering*

Department and *Central Cambridge*.

L_AtomicLocation(*Room* 7, (*Andy's office*, *Supervision Area*, *Office*))
L_NestedLocation(*Floor* 5, (*Meeting Room*, *Room* 7, *Room* 8, *Room* 9, *Room* 10,
Room 11, *Alcove*, *Coffee Area*, *Corridor*))
L_NestedLocation(*Old-LCE*, (*Floor* 4, *Floor* 5))
L_NestedLocation(*Engineering*, (*Old-LCE*, *Fallside*, *GRO*))
L_NestedLocation(*Central Cambridge*, (*Engineering*, *Chemistry*,
Chemical Engineering))

The second library that was used in this experiment contained rules that generate abstract event type definitions, both for the SAL predicates and for the abstract predicates that are used in the example applications; these define the predicates *H_UserIsPresent*, *H_UserCoLocation* and *H_UserInDeducedLocation*. All abstract predicate type definitions can be generated from the AESL definitions of the applications of interest (Chapter 12). The third library used contained rules that generate in SCAFOS the mathematical and logical functions of Chapter 5. A fourth library contained rules that create *trigger-query* nodes such as the ones that are discussed in Chapter 8. This library can be automatically generated by definitions of the applications of interest using the SCALA language (Chapter 12). Finally, a fifth library was used in order to generate the behaviours of both the single-layer and dual-layer knowledge base components (Chapter 6).

11.2 Applications

This section discusses four applications that were written using SCAFOS. Each application defines an abstract event type of interest by providing an AESL definition for that event type through the *Subscribe()* interface of Chapter 8. SCAFOS publishes event instances for that event type. Event correlation is undertaken by SCAFOS' deductive knowledge base component.

Example 1. *"Notify me by email whenever any user moves from one room to another." This application sends an e-mail notification to a requesting user every time any user moves from one room to another in the old LCE. The application subscribes to abstract events of type H_UserInLocation through the Subscribe() interface by providing the following AESL definition:*

$$\begin{aligned} & L\_UserAtPosition(uid, role, (x, y, z)) \\ \wedge & L\_AtomicLocation(rid, rattr, polygon) \\ \wedge & L\_InRegion((x, y, z), rid, rattr) \\ \Rightarrow\ & H\_UserInLocation(uid, role, rid, rattr) \end{aligned}$$

Example 2. *"Notify me by email whenever two users are co-located." This application sends an e-mail notification to a requesting user every time any two users are co-located inside a room in the old LCE. The application subscribes to abstract events of type H_UserCoLocation through the Subscribe() interface by providing the following AESL definition:*

Application	Abstract Events
Example 1	$30,612$
Example 2	$92,870$
Example 3	$30,612$
Example 4	$122,448$

Table 11.1. Abstract event notifications per application in the single-layer architecture.

$$\begin{aligned} & L\_UserAtPosition(uid_1, role_1, (x_1, y_1, z_1)) \\ \wedge & L\_UserAtPosition(uid_2, role_2, (x_2, y_2, z_2)) \\ \wedge & L\_AtomicLocation(rid, rattr, polygon) \\ \wedge & L\_InRegion((x_1, y_1, z_1), rid, rattr) \\ \wedge & L\_InRegion((x_2, y_2, z_2), rid, rattr) \\ \Rightarrow & \textit{H\_UserCoLocation}(uid_1, uid_2, role_1, role_2, rid, rattr) \end{aligned}$$

Example 3. *"Notify me whenever a user is present in any region." This application sends an e-mail notification to a requesting user, whenever any user is present anywhere, irrespective of region. The application subscribes to abstract events of type H_UserIsPresent through the Subscribe() interface by providing the following AESL definition:*

$$\begin{aligned} & L\_UserInLocation(uid, role, rid, rattr) \\ \Rightarrow \quad & H\_UserIsPresent(uid, role) \end{aligned}$$

Example 4. *"Notify me whenever a user is located in Central Cambridge." Whenever a user is "seen" by the location system anywhere in Central Cambridge, SCAFOS undertakes the deduction that the user is contained in all (nested) regions that contain the user's current position. This is denoted through the creation of abstract events of type H_UserInDeducedLocation as defined by the AESL definition that is provided by this example application, as in the previous examples.*

$$\begin{aligned} & L\_UserInLocation(uid, role, rid_1, rattr_1) \\ \wedge & L\_NestedLocation(rid_2, rattr_2, site\text{-}list_2) \\ \wedge & \textit{L\_InList}(rid_1, site\text{-}list_2) \\ \Rightarrow & H\_UserInDeducedLocation(uid, role, rid_2, rattr_2) \end{aligned}$$

11.2.1 Single-Layer Architecture

The above rules were tested in the prototype implementation of SCAFOS discussed in Chapter 10 in a single-layer deductive knowledge base architecture. The behaviour of the deductive knowledge base component is described as follows: for each input event, all facts that were deduced from the user's previous location are retracted and new facts are deduced according to the logic of the example applications discussed above.

The number of abstract event instances that were published by SCAFOS in response to the subscriptions of the applications are presented in Table 11.1. In this case, for the total of the $30,612$ location events received by the Active BAT system, $92,870$ co-locations of pairs of users were calculated, it was deduced $30,612$ times that a user was present and $122,448$ times a higher level containment was deduced from a user's location (all positions within *Old-LCE* also belong to *Central Cambridge*).

Application	Abstract Events
Example 1	425
Example 2	354
Example 3	20
Example 4	80

Table 11.2. Abstract event notifications per application in the dual-layer architecture.

Architecture	Total Abstract Events	Total Processing Time	Average Response Time/ Notification
Dual-layer	879	$142sec/60$ $= 2.367\ min$	$2.367\ min/30612$ $= 7.732 * 10^{-5}\ min$ $= 0.00464\ sec$
Single-layer	276542	$1417\ sec/60$ $= 23.617 \min$	$23.617\ min/30612$ $= 7.714 * 10^{-4}\ min$ $= 0.0463\ sec$

Table 11.3. Performance results.

11.2.2 Dual-Layer Architecture

In the dual-layer knowledge base architecture, only threshold changes in the above predicates trigger the publishing of abstract events. For the application of Example 1, an abstract event of type *H_UserInLocation* is published only when a user moves from one room to another. For the application of Example 2, only the facts that denote changes in a co-location for that user are updated. As a result, if a user changes position within a room where he is co-located with another user without any of the users leaving the room, no additional *H_UserCoLocation* notifications will be generated. For the application of Example 3, the *H_UserIsPresent* predicate will be instantiated once for each user.[1] Similarly, *H_UserInDeducedLocation* abstract event notifications are not generated as long as the user moves within the same floor.

The number of abstract event instances that were generated by SCAFOS using the dual-layer knowledge base architecture are summarised in Table 11.2. This means that during the experiment, 20 users were found to be present, 425 times a user moved from one room to another, 354 pairs of co-located people were observed and since all users moved within a single floor (*Floor 5*) nested locations were deduced only once (four levels of nesting) for each user.

11.2.3 Discussion

The above results provide an indication of SCAFOS' performance. The single-layer application represents the case where *all* input events are considered as threshold events and therefore need to be processed individually by SCAFOS' deductive knowledge base. This corresponds to a heavy processing load for the given application set. The dual-layer architecture represents the case where a relatively small number of input events are considered to be important (threshold events) and therefore contribute to the publication of new abstract event instances.

SCAFOS' behaviour is evaluated with the help of two metrics: the *number of published abstract event instances* and the *average response time per notification*. The latter denotes the average time needed by SCAFOS for processing each input event (see Table 11.3). The average response time is

[1] Although it is possible to denote the absence of the user, e.g., by the fact that there are no sightings of that user for a long period of time, this was not a requirement of this experiment. Hence, the *H_UserIsPresent* predicate is not retracted when a user leaves the old LCE.

calculated from the total processing time, averaged over the number of input events (30612). The total processing time is calculated as the amount of time required by SCAFOS' deductive component for pattern-matching all the input events. Note that the total processing time does not include the cost associated with tasks related to event publishing such as marshalling arguments for transmission over the network. Both architectures are compared on the basis of these metrics. The overall performance results for the applications discussed in this chapter confirm the theoretical analysis presented in this dissertation by demonstrating the following:

- The dual-layer knowledge base architecture is 10 times more efficient than the single-layer architecture in terms of processing time and approximately 300 times more efficient in terms of the number of published abstract events (Table 11.3).
- For the worst-case scenario of the single-layer architecture, the average response time per published event is still small enough to accommodate the measured upper bound on the event rate generated by SPIRIT, $0.12\ sec.$ These results are satisfactory for the office scenario; however, for very large-scale implementations, there is scope for further testing. Such testing will investigate any event loss that may occur if the input rate exceeds SCAFOS' processing rate. It is worth noting that losing low-level input events can be tolerated as long as these are redundant - i.e., mere confirmations of the same abstract state.

11.3 Examples

This section presents a set of applications that use multiple integrated SCAFOS components. These are given here for illustration purposes.

- Whenever I am in the same room as Pablo, remind me to return his book.
- If I am walking, teleport my desktop to the closest empty meeting room. If I am sitting down, teleport my desktop to the closest PC.
- If I am walking, forward my email on my phone by SMS. If I am sitting at a PC, forward any notifications to my email account. If I am located within any meeting-room, withhold any notifications.
- Notify me by email when a systems administrator or a C++ expert is in the kitchen.
- Set up a conference call among the locations that correspond to a meeting room that is the closest meeting room to each director of each department in the University of Cambridge, among those that have been empty for at least ten minutes.
- If any user approaches the coffee-machine, then approaches the sink and then approaches the coffee-machine again without leaving the room in the interim, notify all subscribers for coffee notifications that fresh coffee is being brewed.

Decision-Making Applications

- If the probability that someone will make coffee in the next hour is higher than 50%, subscribe me for notifications about the status of the coffee.
- Calculate the requirements in food supplies for meetings for the coming week (assuming a given quantity of coffee and cookies per meeting) only considering meetings that may happen with a probability higher than 60% and confidence level higher than 80%.

System Configuration Applications

- Whenever a user is walking in the corridor, stop monitoring this user until the user has exited the corridor.

Queries

- Where is my supervisor most likely to be between 10 am and 11 am today?
- When is the meeting room most likely to be empty?

SCAFOS' current prototype implementation supports most of the above applications (see Chapter 10). The above queries and decision-making applications are implemented using real historical data, originally derived from SPIRIT. System configuration applications are not currently implemented, but their implementation should be straightforward using SCAFOS and standard engineering practice.

11.4 Conclusions

This chapter discusses an evaluation of SCAFOS' performance in a real scenario. Four applications were run for a period of over 40 hours during which SCAFOS' performance was tested. Two different architectures of the deductive knowledge base component were tested and compared. The results demonstrate that the single-layer architecture is sufficient for the given application set when evaluated in the old LCE environment. Furthermore, the dual-layer architecture is substantially more efficient than the single-layer architecture, both in terms of the processing load as well as in terms of the number of published event instances.

Chapter 12

Sentient Computing Applications LAnguage: SCALA

This chapter describes the Sentient Computing Applications LAnguage (SCALA). The SCALA language is an XML-based language that has two principal goals.

- To create specifications for context-aware applications by binding representations of available actions to abstract event definitions. Phrases in SCALA are compiled into an *implementation process* similar to an execution thread that undertakes the end-to-end development of context-aware applications using SCAFOS.
- To create and deploy SCAFOS and its components. This includes creating the SCAFOS components (Chapter 10) as well as creating specifications of the adaptive behaviour of components of the programmable infrastructure including the location system and its middleware components.

12.1 The Anatomy of SCALA

SCALA consists of three sublanguages and an API that are used to implement the above scheme:

1. The Abstract Event Specification Language (AESL).
2. The Abstract Event Filter Specification Language (AEFSL).
3. The ECA Application Specification Language (ECAAS).
4. The SCAFOS support API.

AESL is a language for creating definitions for abstract predicates and subscribing to changes in the values of these predicates - in other words, abstract events. The filtering language AEFSL is used alongside AESL to restrict the selection of the instances of a specific abstract predicate according to the conditions of the filter. ECAAS is a language for creating context-aware application specifications by binding abstract predicates to specific *action predicates* that represent actions available in the environment.

AESL and AEFSL are based on temporal first-order logic (TFOL), and they implement a subset of its operators. We refer to these as separate languages because each offers different functionality and are therefore restricted in terms of what the user can do with them. ECAAS is an ECA-based language, i.e., its statements consist of a conditional LHS part and an action (RHS) part. The language support for building the SCAFOS framework and controlling the adaptive behaviour of its components consists of a set of interfaces (API) and is organised in modules (libraries). This is discussed in detail in Section 12.5.

Using the above languages, the creation of context-aware applications using SCALA entails the following steps.

1. Create an ECAAS statement.
2. Create an AESL definition (including a type definition).
3. Create a filter (AEFSL definition).

The ECAAS statement is compiled into a context-aware application that subscribes to the AED Service by providing the AESL definition and the filter. It then listens for notifications of abstract events of interest. On receipt of such an event, the action specified in the RHS of the ECAAS statement is triggered by SCAFOS.

12.2 Design Principles

- Phrases in AESL and AEFSL are translated into a thread that invokes SCAFOS APIs and executes deductive knowledge base queries. For this reason, the underlying driver behind SCALA's design principles has been implementation efficiency and query optimisation (see Chapter 7). This is discussed in more detail in the section where each language is presented.
- SCALA has been implemented as an XML language as XML is an open, widely used, powerful schema for distributed systems. Since SCALA is inherently heterogeneous and comprises of three sublanguages, XML's semantic tagging provides a natural separation between these languages, thus facilitating parsing.
- SCALA, "ladder" in greek, is an appropriate representation of the synthetic approach taken in AESL, where each abstract predicate is synthesised from other abstract and concrete predicates.

12.3 Abstract Event Definition Language (AESL)

AESL is a language for making abstract event definitions using rules. Rules contain negation, existential quantification and universal quantification. AESL uses a specific syntax: *an abstract event definition* (AESL def.) consists of one or more *implications* with all their variables free and a single conclusion. In case of only one implication, the LHS is an AESL formula and the RHS is the abstract predicate of interest. In case of more than one rule the RHS of the last rule is the abstract predicate of interest (target predicate), while the RHS of each intermediate rule is an intermediate abstract predicate.

Example 1. *Locate the closest location to each user among the locations that have been empty for at least 5 min.*

$$
\begin{aligned}
&(\nexists u\ UL(u, rid, role, rattr) \wedge AL(rid, rattr, polygon) \\
&\Rightarrow EL(rid, rattr, t_1)) \\
&D(v_1, u, role, rid_2, rattr_2) > D(v_2, u, role, rid_1, rattr_1) \\
&\Rightarrow CL(u, role, rid_1, rattr_1, t_2) \\
&CL(u, role, rid_1, rattr_1, t_2) \wedge EL(u, role, rid, rattr, t_1) \\
&\wedge |EL, CL|_{t>300} \\
&\Rightarrow CEL(u, role, rid, rattr, t_3)
\end{aligned} \tag{12.1}
$$

Note that t_1, t_2 and t_3 is the time of generation for EL, CL and CEL respectively. The abstract event detector for (12.1) is portrayed in Figure 12.1.

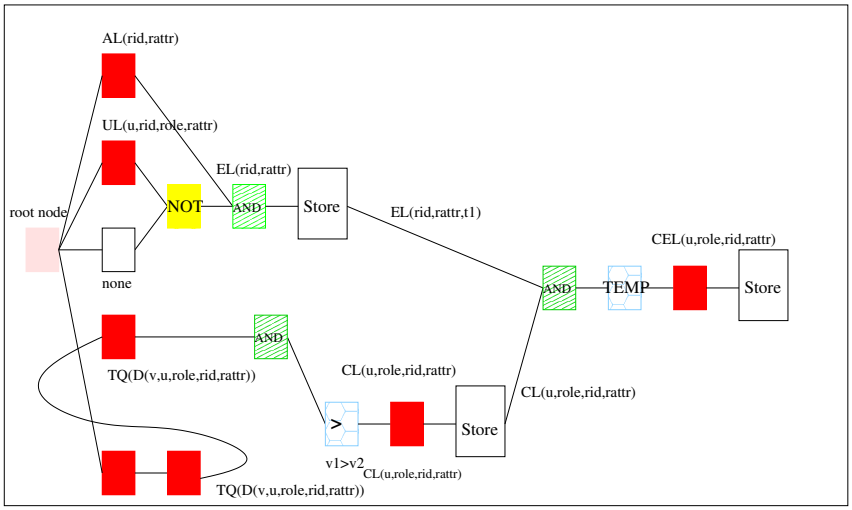


Figure 12.1. An abstract event detector for Equation (12.1).

Type Definition. AESL is a typed language. Type definitions are therefore necessary whenever a new abstract predicate is defined. Data definition statements are compiled into *deftemplate* statements in the deductive knowledge base component. Until a type definition has been provided for an abstract predicate, this predicate will not be usable in an AESL specification. If a prior definition for the predicate of interest already exists in the knowledge base, a type definition statement that attempts to redefine the template is ignored, and an error is returned to the user.

For example, the following type definition statement can be used to define the abstract predicate $H\_EmptyLocation(EL)$:

$$H\_EmptyLocation(rid\ string)(rattr\ string)$$

12.3.1 Temporal Reasoning.

Local "now". In each deductive knowledge base component, the instant when a primitive event (produced by the sensor infrastructure) is asserted into SAL defines the current moment in time, locally, in that sensor-driven component. As the reception of each event causes the re-calculation of all the constraints in the knowledge base, the global state for that component is updated accordingly. If we assume that the propagation takes time dt, then "now" is $t + dt$.

This means that the publisher's local "now" depends on the following three factors:

- The latency between the actual occurence of the event, (e.g., user movement) and the generation of the primitive event that reflects this.
- The latency between the generation of events and the reception of those events by the Deductive KB component.
- The latency between the reception of an event by the Deductive KB component and the update of all the abstract predicates that depend on this event, i.e., the creation of the abstract event of interest.

The first factor depends directly on the sampling efficiency of the sensor technology. The Active BAT is very efficient in this aspect, as it supports very high sampling rates that are variable and can adapt to the speed of the physical object. Such rates allow the classification and recognition of human movements

Operator	Description
$e_1; e_2$	e_1 before e_2
$e_1; e_2 ! e_3$	e_1 before e_2 without e_3 in between
$\|e_1, e_2\|_{T=t_1}$	e_2 happened within time t_1 from e_1.
now (implicit)	At the current instant.
timestamp	At time t.

Table 12.1. AESL temporal operators.

based on a small sample of location data. So this latency is, in reality, very small and can be ignored. The second latency is also very small, as the event sources and the deductive knowledge base component are local. A high-speed local network can be used. Furthermore, SPIRIT is inherently scalable, using zone manager entities to gather sightings from the sensors locally. The third latency is addressed by the proposed layered architecture of the deductive KB component, which is restricted only by the amount of memory; aggressive garbage collection of historical data can be used to maximize the available memory. A fast machine can be used to increase processing speed, and pattern matching is inherently fast and efficient based on the Rete algorithm.

Temporal Operators. AESL supports the temporal operators of Table 12.1, which have been inspired by the results presented in [63, 81]. The operators can be combined to form expressions. The operator "now" is implicit. The variables e_1, e_2, e_3 are instances of the state predicates in the model, e.g., UL.

In a distributed sensor-driven system there is no global time nor can an upper bound be guaranteed on event transmission delays. Because of factors like clock skew, it is sometimes impossible to say whether $e_1; e_2$ is true or false. An ordering convention may be imposed, based naïvely on locally generated event timestamps. This may be appropriate for some applications. However, this is not sufficient for the needs of context-aware applications that depend on the correct *causal* order of the abstract events that synthesise the abstract knowledge of interest. If an application requires strictly correct sequencing and can afford to discard ambiguous cases, then interval timestamps can be used for events as proposed in [58] and used in [81]. In the latter, the application is made aware of the cases where partial order is ambiguous, and it is offered weak or strong sequencing in order to handle the ambiguity. For cases where ambiguity cannot be tolerated, alternative methodologies need to be adopted.

12.3.2 BNF

The AESL syntax is described next using the Backus Naur Form (BNF).

$$
\begin{aligned}
AESLDef &:= [Rule];\\
Rule &:= Sentence \Rightarrow Abstract\ Predicate \mid Binding;\\
Binding &:= pred\_var < -Predicate\ (Term);\\
Sentence &:= Atomic\ Sentence \mid Temp\ Sentence \mid Sentence\ Conn\ Sentence \mid\\
&\quad Quantifier\ Varlist\ Sentence \mid \neg Sentence \mid (Sentence)\\
Temp\ Sentence &:= pred\_var\ TempOp\ pred\_var \mid pred\_var \mid pred\_var\ timestamp\\
&\quad \mid Temp\ Sentence\ TempOp\ temp\ Sentence;\\
TempOp &:= ; \mid !;\\
Term &:= Function\ (Termlist) \mid mathitConst \mid Var;\\
Conn &:= \wedge \mid \vee \mid = \mid < \mid >;
\end{aligned}
$$

Operator	**Description**
$c_1 \wedge c_2$	$F_1 \cap F_2$
$c_1 \vee c_2$	$F_1 \cup F_2$
$\neg c_1$	$E - F_1$
$(c_1 \vee c_2) \wedge c_3 \Leftrightarrow (c_1 \wedge c_3) \vee (c_2 \wedge c_3)$	$(F_1 \cap F_3) \cup (F_2 \cap F_3)$
$c_1 \vee c_2 \wedge c_3 \Leftrightarrow c_1 \vee (c_2 \wedge c_3)$	$F_1 \cup (F_2 \cap F_3)$

Table 12.2. Filter algebra operators.

12.3.3 Abstract Event Filter Definition Language

As mentioned in the previous section, AESL definitions do not contain constants. Therefore, each AESL definition, when compiled, leads to the generation of an abstract event detector that detects changes to *all* instances of an abstract predicate. In addition to the AESL definitions, it is possible for the user to restrict even further the selection of the instances of the abstract predicates of interest by using *filtering*. Filtering is equivalent to selecting a subset of instances of a specific predicate by specifying a set of attributes and constraints on these attributes. Each attribute constraint is a tuple specifying a name, a binary predicate operator and a value for an attribute. An attribute $a = (name_a, value_a)$ matches an attribute constraint $\phi = (name_\phi, operator_\phi, value_\phi)$ if and only if $name_a = name_\phi \wedge operator_\phi(value_\alpha, value_\phi)$. We say that an attribute α satisfies or matches an attribute constraint ϕ with the notation $\alpha \prec \phi$.

We define the Abstract Event Filter Definition Language (AEFSL) as a language for specifying filters. Attribute constraints can be connected with OR, NOT and AND operators. When a filter is used in a subscription, all conjoined constraints must be matched. Disjunction is equivalent to applying multiple filters, one for each disjoined condition. A negated constraint is equivalent to selecting all attributes whose values do not match the one specified in the constraint.

A filter can also be defined as a set of predicate instances whose attributes a_i match the filter constraints as explained above. In this way, the conjunction, disjunction and negation of attribute constraints can be defined in terms of set algebra. Table 12.2 summarises this. In Table 12.2 c_1, c_2 are conditions of the filter and F_1, F_2 are the sets that the conditions correspond to, respectively. Finally E is the set of all predicates.

12.3.4 Filters

Filters are linear abstract event detectors. Filters are restricted to one-input nodes, which makes them economical; in particular, they do not have any two-input nodes. Each attribute constraint contained in the AEFSL definition is compiled into a query node. A filter for selecting only the instances of the $H\_UserInLocation(Ek236, role, Room\ 5, rattr)$ predicate that correspond to user ek236 being in Room 5 is portrayed in Figure 12.2. Node n_2 selects the instances where the user id equals Ek236 and node n_3 selects the instances where the region id equals Room 5.

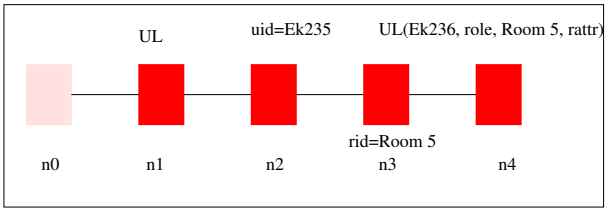
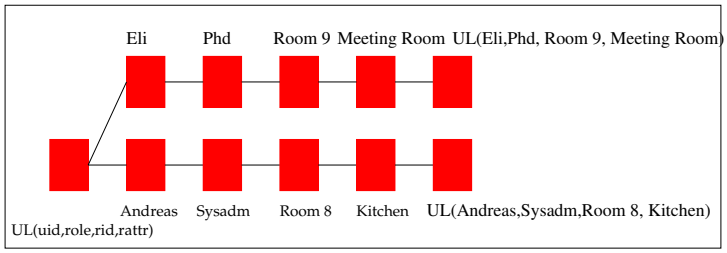


Figure 12.2. A filter.

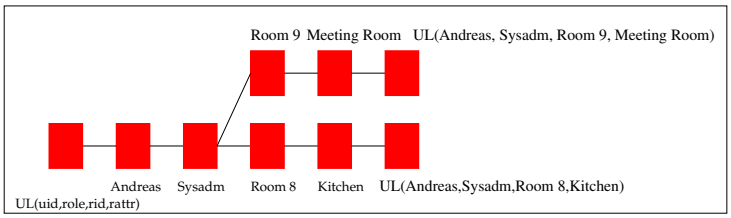
Filters have the property that they do not replicate computation. Attribute constraints that are com-

mon between filters are compiled into a single query node that is shared between the filters. Figure 12.3(b) portrays the combination of the filters of Equation 12.2. Filters that do not share nodes ((12.5) and (12.7)) are guaranteed to have different attribute constraints and therefore no replication is possible in that case (Figure 12.3(a)).

$$\begin{aligned} &\textit{uid=Andreas} \wedge \textit{role=Sysadmin} \wedge \textit{rid=Room 8} \wedge \textit{rattr= Kitchen} \\ &\textit{uid=Andreas} \wedge \textit{role=Sysadmin} \wedge \textit{rid=Room 9} \wedge \textit{rattr=Meeting Room} \end{aligned} \tag{12.2}$$



(a)



(b)

Figure 12.3. Filter combination.

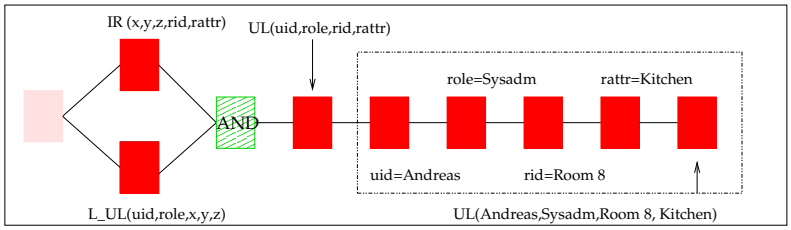
12.3.5 BNF

$$\begin{aligned} \textit{Filter} &:= \textit{Atomic Sentence} | \textit{Filter Conn Filter} | \neg \textit{Filter} | (\textit{Filter}) \\ \textit{Atomic Sentence} &:= \textit{Term} = \textit{Term} | \textit{Term} > \textit{Term} | \textit{Term} < \textit{Term}; \\ \textit{Term} &:= \textit{Constant} | \textit{Variable}; \\ \textit{Conn} &:= \wedge | \vee; \\ \textit{Const} &:= \textit{Str}; \\ \textit{Var} &:= \textit{Str}; \end{aligned}$$

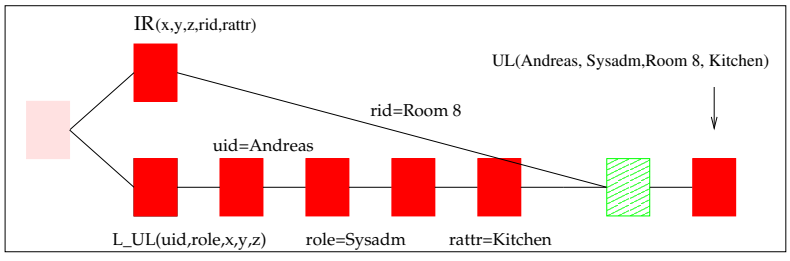
12.3.6 AESL and AEFSL Design Principles

Query Optimisation. Because Abstract Event Detectors are implemented as Rete networks, which constitute the reasoning structures in a deductive knowledge base, they can be regarded as knowledge base *queries* (Chapter 6). Similarly to database queries, abstract event detectors can be optimised with respect to the computational complexity required for abstract event detection. The syntax of the AESL definitions (in successive TFOL implications, each synthesising the goal abstract predicate) leads to the generation of abstract event detectors that are computationally *optimised*. An alternative structure, which would consist of a single implication, leads to a worst-case abstract-event detector (Chapter 7).

Avoiding Duplication of Computations. Although filtering could be integrated into AESL by regarding filter constraints as additional TFOL conditions, abstract event definition and filtering has been kept separate. This has been done for implementation efficiency, so that overlapping or duplicate event definitions and filters do not cause replication of computation. Two principles were adopted, firstly, each AESL definition leads to the creation of a single abstract predicate. This requires that any filters that are applied subsequently contain constraints on the attributes of a single predicate and therefore have linear computational complexity and are, in this way, relatively inexpensive with regard to the computational cost associated with the detection. Secondly, users are encouraged to write AESL definitions that use variables as arguments for predicates rather than constants. This ensures that the deduction of *all instances* of the abstract predicate, which is computationally expensive, is performed once, and a subset of instances of the predicate are selected later on, by filters. Because filters can be combined, they do not replicate computation.



(a)



(b)

Figure 12.4. Unrestricted vs. restricted abstract predicates in terms of their attribute values.

Example. The query that corresponds to the phrase "Somebody is somewhere" is mapped to the following AESL specification:

$$L\_UL(uid, role, x, y, z) \wedge IR(x, y, z, rid, rattr) \Rightarrow UL(uid, role, rid, rattr) \quad (12.3)$$

The predicate $UL(Andreas, Sysadm, Room\ 8, Kitchen)$ which corresponds to *"Andreas, the system administrator is in Room 8 which is a kitchen"* can be generated either directly by the specification:

$$L\_UL(Andreas, Sysadm, x, y, z) \wedge IR(x, y, z, Room\ 8, Kitchen) \Rightarrow$$
$$UL(Andreas, Sysadm, Room\ 8, Kitchen) \quad (12.4)$$

or it can be derived from (12.3) by applying a filter:

$$uid = Andreas \wedge role = Sysadm \wedge rid = Room\ 8 \wedge rattr = Kitchen \quad (12.5)$$

Equation (12.3) combined with (12.5) is implemented as shown in Figure 12.4(a), while (12.4) is implemented as portrayed in Fig. 12.4(b). Figure 12.5 portrays the combination of (12.4) with a query that corresponds to the phrase *"Eli, who is a PhD student, is in Room 9, which is a meeting room"*, as defined in (12.6).

$$L\_UL(Eli, Phd, x, y, z) \wedge IR(x, y, z, Room\ 9, Meeting\ Room) \Rightarrow$$
$$UL(Eli, Phd, Room\ 9, Meeting\ Room) \quad (12.6)$$

If the filter of (12.7) was used instead, combined with Equation (12.3) and the filter of (12.2), then the structure of Figure 12.6 would be generated as a result. It is easy to see that the structure in Figure 12.5 calculates overlapping sets of predicate instances whereas the structure in Figure 12.6 does not.

$$uid = Eli \wedge role = Phd \wedge rid = Room\ 9 \wedge rattr = Meeting\ Room \quad (12.7)$$

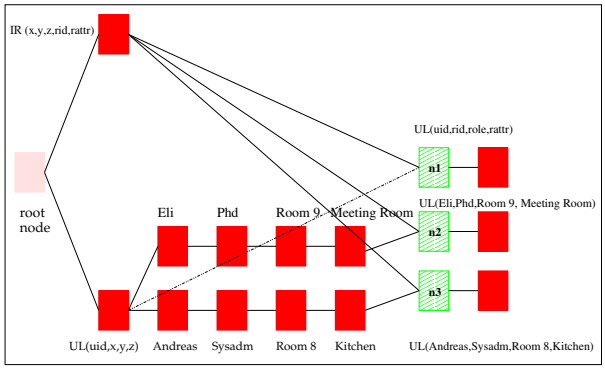


Figure 12.5. Replication of computational resources with restricted predicates.

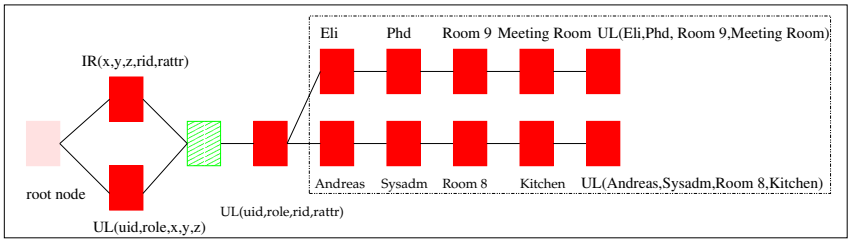


Figure 12.6. Avoiding replication of computation by using un-restricted predicates.

LCE Action Predicates	Description
RemoteDesktop(userID, hostname)	Transfer the user's virtual desktop to a remote host.
SubscribeToCoffeeNotifications(ActiveBATId)	The user receives an audible notification whenever a fresh pot of coffee is available.
SMS(addresss)	Send an sms mesage to the specified address.
ScanToEmail(address)	Send a scanned image to an email address.
BATSleep(ActiveBATId)	Put BAT in quiet mode.
KillAgent(AgentId)	Kill agent that has violated a security rule.
SetMonitoringRate(ActiveBATId,value)	Set the sampling rate for a BAT to the specified value.
DigitalPhotoToBroadbandPhone(ActiveBATId)	Send a digital image to a user's broadband phone.
Load/Unload OUIJA object	Loads/Unloads OUIJA objects from the cache, according to the context of the entities that correspond to these objects.

Table 12.3. LCE action predicates.

12.3.7 Temporal Operators

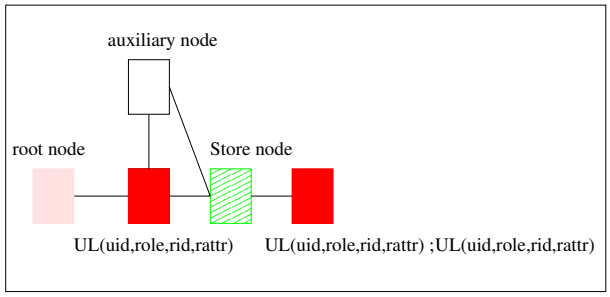
Figures 12.7 and 12.8 portray the implementation of three common AESL temporal operators. As can be seen from these graphs, their implementation is straightforward.

12.4 Event-Condition-Action Application Specification Language

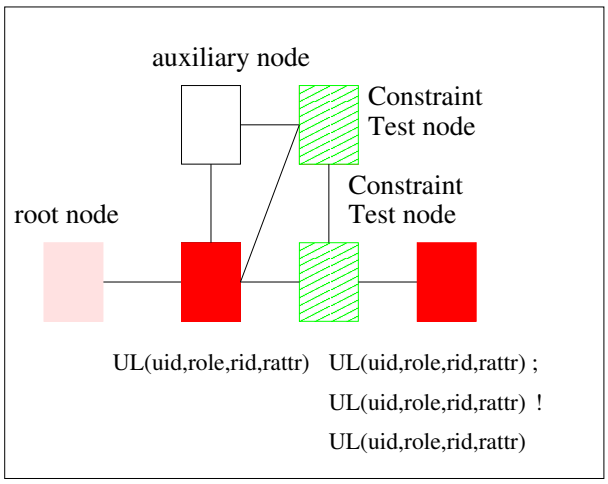
The ECAAS language is an ECA-like language for writing specifications of context-aware applications based on FOL. Each context-aware application is written as an ECA statement consisting of a conditional (LHS) and an action part (RHS) part. The conditional part is compiled into an abstract predicate name. The RHS is compiled into a set of Jess *Userfunction* calls. Actions are specific to each sensor-driven component, and they are determined by the available applications. For example, in the LCE, the actions to be used in ECAAS statements belong to three broad categories: actions that offer context-aware functionality, actions that are taken in response to security violations (e.g., agent "killing") and those that involve system actions, such as modifying the Active BAT monitoring rate and changing the page replacement algorithm in SPIRIT's object cache [103]. Table 12.3 summarises some of the available action predicates in the LCE.

Two tags, the *AdminSpec* and *Else* tags, are used in order to control the newly developed application. The *AdminSpec* tag controls the scope of the created context-aware applications and the period for which historical data is kept. The *Else* tag is used to determine the desired action while the conditions do not hold.

Example. Consider the following context-aware application. *"Whenever I am in an empty meeting room, teleport my desktop to the EC in that room. Else, teleport my desktop to my workstation (Budder.eng.cam.ac.uk)."* The full XML definition for this application is shown below. *UEL* stands for



(a) $UL(uid, role, rid, rattr); UL(uid, role, rid, rattr)$



(b) $UL(uid, role, rid, rattr); UL(uid, role, rid, rattr)!UL(uid, role, rid, rattr)$

Figure 12.7. Temporal Rete network operators.

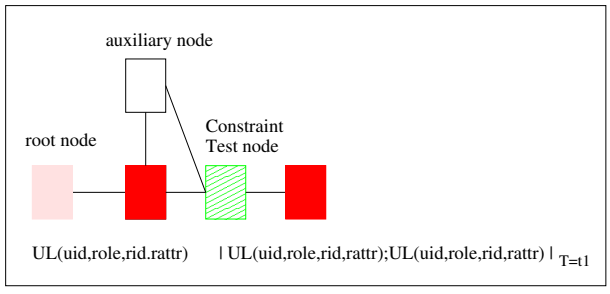


Figure 12.8. $|UL(uid, role, rid, rattr); UL(uid, role, rid, ratrr)|_{T=t}$

$H\_UserInEmptyLocation(uid, role, rid, rattr)$.

$< CreateECAApp >$
$< ECAAppSpec >$
$< RuleExp >$
$< AbsPredicate >$ `UEL(Ek236, role, rid, Meeting Room)` $< /AbsPredicate >$
$< ActionPredicate >$ `RemoteDesktop(Ek236, rid)` $< /ActionPredicate >$
$< Else >$
$< ActionPredicate >$ `RemoteDesktop(Ek236, Budweiser.eng.cam.ac.uk)`
$< /ActionPredicate >$
$< /Else >$
$< /RuleExp >$
$< AbsPredDef >$ `EL`
$< Param >$ *room name* `string` $< /Param >$
$< Param >$ *rattr* `string` $< /Param >$
$< /AbsPredDef >$
$< AdminSpec >$
$< Scope >$ *forever* $< /Scope >$
$< /AdminSpec >$
$< /ECAAppSpec >$

$< AESLDef >$
$< Sentence >$ $\nexists$`uid` UL`(uid, rid, role, rattr)`
$\Rightarrow$ `EL(rid, rattr)`
$\forall$`EL(rid, rattr); UL(uid, rid, rattr, role)`
$\Rightarrow$ `UEL(uid, rid, role, rattr)`
$< /Sentence >$
$< /AESLDef >$
$< /AEFSLDef >$ `rattr = Meeting Room and uid = Ek236` $< /AEFSLDef >$
$< /CreateECAApp >$

The above XML definition is passed into the AED Service and compiled into the abstract event detector of Figure 12.9, which is placed at each sensor-driven component. Whenever an abstract event of type $UEL(Ek236, role, rid, MeetingRoom, activate)$ is detected, the remote desktop of user Ek236 is teleported into the Meeting Room that matches the specification. Else, the remote desktop is transported to Budweiser.eng.cam.ac.uk.

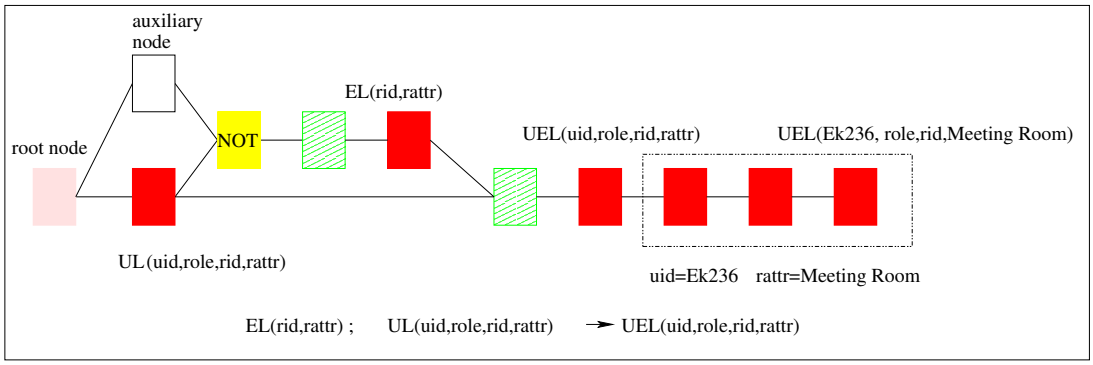


Figure 12.9. H_UserInEmptyLocation(Ek236, role,rid,Meeting Room).

12.4.1 SCALA DTD

The DTD for SCALA is portrayed below. The elements $AESLDef$ and $AEFSLDef$, are defined according to the respective BNF syntax in Section 12.3.2 and Section 12.3.5 respectively.

$$
\begin{aligned}
&CreateECAApp(ECAAppSpec, AESLDef, AEFSLDef)\\
&ECAAppSpec(RuleExp+, AbsPredDef, AdminSpec*)\\
&RuleExp(AbsPredicate\,ActionPredicate+, Else)\\
&AbsPredicate(\#PCDATA)\\
&ActionPredicate(\#PCDATA)\\
&Else(ActionPredicate+)\\
&AbsPredDef(\#PCDATA, paramlist*)\\
&paramlist(\text{“(“ } Param, \#PCDATA, \text{“)”}+)\\
&AdminSpec(history*, Scope*)\\
&History(\#PCDATA)\\
&Scope(\text{“once”}|\text{”forever”}|time)\\
&time(\#PCDATA)
\end{aligned}
\tag{12.8}
$$

The $AESLDef$ and $AEFSLDef$ statements are defined according to the BNF notation of Section 12.3.2 and Section 12.3.5, respectively.

12.5 SCALA SCAFOS Support

The SCALA statements for creating the infrastructure are contained in libraries (modules). SCALA supports the modules of Table 12.4. This includes two specialised modules for connecting with the SPIRIT and QoSDREAM systems, using them as primitive context sources. Each module is described in more detail in Appendix B. The overall functionality of SCALA is shown in Figure 12.10.

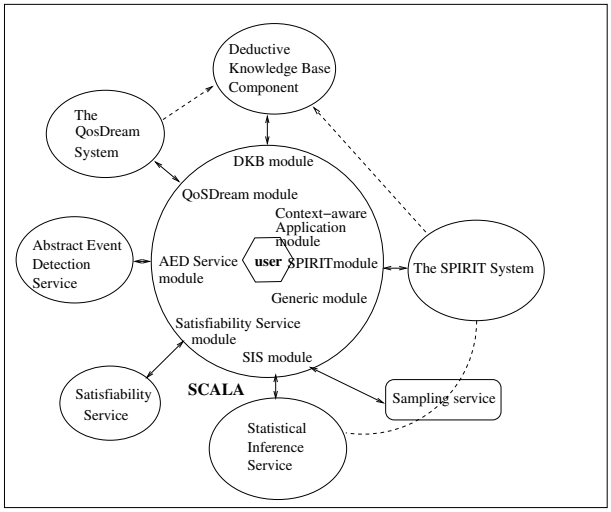


Figure 12.10. The SCALA language architecture

Module name	**Description**
Context-Aware Application Module	APIs for developing applications.
Deductive Knowledge Base Module (DKB)	APIs for creating and controlling Deductive Knowledge Base components.
Abstract Event Detection (AED) Service Module	APIs for creating and controlling the AED service.
SPIRIT and Active BAT Module	APIs for linking the infrastructure with an existing SPIRIT system. APIs for controlling the performance of the SPIRIT object model. APIs for controlling the ActiveBAT system.
Satisfiability Service Module	APIs for creating and controlling the Satisfiability Service.
Statistical Inference Service (SIS) Module	APIs for creating and controlling the Statistical Inference Service Module
QosDREAM Module[87]	APIs for linking the infrastructure to an existing QoSDREAM system.
Generic Module	APIs for linking legacy sensor technologies with the SCAFOS framework.

Table 12.4. SCALA Modules

Chapter 13

Conclusions and Further Work

The thesis of this dissertation, stated in Chapter 1, and discussed, expanded and demonstrated through the body of this dissertation, is that it is both necessary and beneficial to provide a framework for context-awareness in sensor-driven systems. More specifically, this dissertation described the design and implementation of the SCAFOS framework. SCAFOS has two novel aspects. Firstly, it provides powerful tools for inferring *abstract knowledge* from low-level, *concrete* knowledge, verifying its correctness and estimating its likelihood. Such tools include Hidden Markov Models, a Bayesian Classifier, abstract events defined in terms of Temporal First-Order Logic, the theorem prover SPASS and the production system CLIPS. Secondly, SCAFOS provides support for simple application development through the XML-based SCALA language. By introducing the new concept of a generalised event, an *abstract event*, defined as a notification of changes in abstract system state, expressiveness compatible with human intuition is achieved when using SCALA. The applications that are created through SCALA are automatically integrated and operate seamlessly in the various heterogeneous components of the context-aware environment even while the user is mobile or when new entities or other applications are added or removed in SCAFOS.

To the best of the author's knowledge, SCAFOS represents the first system to model context-awareness in sensor-driven systems.

13.1 Contributions

This thesis and the resulting SCAFOS framework has made several important contributions in the field of context-awareness. In summary, this thesis has resulted to the following artifacts:

- A framework called SCAFOS, which provides the following features:
  - It implements a state-based, formal model for context-awareness in sensor-driven systems that integrates knowledge while maintaining knowledge integrity. A *state-based* representation is a novel way of modelling distributed systems, and it is used instead of traditional-event based models that are insufficient for sensor driven systems.
  - Knowledge in the model is maintained in a dual-layer knowledge base. The lower layer maintains concrete knowledge predicates, e.g., it knows of the position of a user in space, in terms of his coordinates *x,y,z*. The higher layer maintains abstract knowledge predicates about current and historical states of the Sentient Environment along with temporal properties such as the time of occurrence and their duration, e.g., it knows of the room a user is in and for how long he has been there. Abstract predicates change much less frequently than concrete predicates, namely, only when certain threshold events happen. Knowledge is

retrieved mainly by accessing the higher layer, which entails a significantly lower computational cost than accessing the lower layer. In this way this scheme acts as a dual-layer cache for knowledge predicates.

- SCAFOS introduces the novel concept of *abstract events*. An abstract event is a generalised event defined as a change in the value of an abstract state predicate. SCAFOS introduces the AESL language for defining new abstract events from concrete and abstract knowledge predicates. AESL is a TFOL-based language. Phrases in this language can capture human intuition about systems state that was not definable before. Such phrases involve negation, (e.g., *empty* room) and semantically transparent reasoning with global state, (e.g., locate the closest empty meeting room) even when this refers to multiple distributed sensor-driven components with different entities. Furthermore, the AESL language is designed with implementation efficiency in mind. AESL phrases are compiled into reasoning structures called abstract event detectors, which are optimised for computational efficiency. Abstract event detectors are implemented as Rete networks. AESL is complemented by the filtering language AEFSL.
- SCAFOS is dynamically extensible. *Dynamic extensibility* refers to the ability to modify the modelled physical entities and create or remove applications without taking the sensor-driven system offline and without having to recompile existing applications.
- SCAFOS contains the SCALA language for using the SCAFOS framework. SCALA is an XML-based language that contains the following three sublanguages: the AESL language, the AEFSL language and the ECAAS language. The ECAAS language is an extended ECA language for building applications easily, by binding AESL definitions to action predicates that represent actions available in the environment.
- SCAFOS contains powerful tools for extending the model with abstract knowledge by means of probabilistic statistical inference, using *Hidden Markov Models (HMMs)* and *Bayesian prediction*. More specifically, SCAFOS contains an HMM-based scheme for detecting and recognising human movements from position streams. This system is independent of user and domain. SCAFOS also contains a scheme for estimating the likelihood of future concrete or abstract predicates being true. This has a number of benefits since it enables decision making in the absence of knowledge sources.
- SCAFOS contains tools for checking the correctness of user requirements and their compatibility with the models of the distributed sensor-driven components.

- An implementation of SCAFOS, based on CORBA, consisting of the following services:
  - A Context-Aware Application Service that enables the simple development of context-aware applications with little programming overhead.
  - A Statistical Inference Service that detects human movement from location data and estimates the likelihood that an instance of either concrete or abstract knowledge will hold in the future.
  - An Abstract Event Detection Service that detects changes of state in the entities in the model and translates them into abstract events.
  - Abstract Event Detectors are implemented as Rete networks that are structured as a deductive knowledge base. The deductive knowledge base can be layered in order to support scalable abstract reasoning (Deductive Knowledge Base component.)
  - A Satisfiability Service that proves the correctness of user requirements and their semantic compatibility with the model stored in each distributed sensor-driven system component.

The results of this thesis have broad implications both for users and the developers of context-aware applications. By using the SCAFOS framework, a user can easily develop context-aware applications that are tailored to the user's needs. Developers of context-aware applications can use the SCAFOS model in order to develop context-aware applications that are integrated automatically with existing applications and operate seamlessly with heterogeneous, sensor-driven components. All applications remain operational even when the underlying framework is extended dynamically with new entities, such as people and regions. Finally, the SCAFOS model can be seen as a standard against which the pros and cons of each context-aware platform can be measured. This promotes commercialisation both for context-aware applications and for supporting platforms.

13.2 Future Work

The work described in this dissertation constitutes a first step into the emerging field of modelling context-awareness, and, as such, it is open to improvements in order to increase its functionality, ease of use and to reduce its operating overhead. Interesting directions for extending this work include integrating SCAFOS with policy-based systems for access control, user authentication and user privileges. A conflict resolution scheme based on the the work presented in Chapter 9 is essential, in order to resolve not only possible conflicts between the functionality of different applications but also in the policies that govern the behaviour of SCAFOS in heterogeneous environments.

Integrating SCAFOS with a distributed, peer-to-peer network architecture and implementing some of the core operations of SCAFOS such as distributed abstract event detection (Chapter 8) using mobile code is another promising extension to this work. Using mobile code, such as mobile agents, Abstract Event Detectors could migrate automatically to nodes in the network where they can be optimally processed, i.e., closer to the event sources.

Finally, SCAFOS can be applied to modelling sensor-networks.

Appendix A

On the Implementation of Transparent Reasoning with Distributed State using Finite Automata

A.1 Definitions

Definition 1. *By predicate or a Boolean-valued function on a set S we mean a total function P on S such that for each $a \in S$ either*

$$P(a) = true, \;\; or \;\; P(a) = false$$

where true and false are a pair of distinct objects called truth values.

We define the predicate $P(x_1, \cdots, x_m)$ as an $(m - ary)$ predicate on S^m.

Definition 2. *Let $P(t, x_1, \cdots, x_n)$ be an $(n + 1) - ary$ predicate. Consider the predicate $Q(y, x_1, \cdots, x_n)$ defined by*

$$Q(y, x_1, \cdots, x_n) \Leftrightarrow P(0, x_1, \cdots, x_n) \vee P(1, x_1, \cdots, x_n) \vee \cdots \vee P(y, x_1, \cdots, x_n).$$

Thus the predicate $Q(y, x_1, \cdots, x_n)$ is true only in the case where there exists a value of $t \in S$ such that $P(t, x_1, \cdots, x_n)$ is true. Q can be written as:

$$(\exists t)_{\leq y} P(t, x_1, \cdots, x_n).$$

The expression $(\exists t)_{\leq y}$ is called a bounded existential quantifier. In the situation where there exists a value of $t \in \mathbb{N}$ such that $P(t, x_1, \cdots, x_n)$ is true, we write:

$$(\exists t) P(t, x_1, \cdots, x_n).$$

The expression $(\exists t)$ is called an existential quantifier.

Definition 3. *The expression $(\forall t)$ is called a universal quantifier.*

$$(\forall t) \; is \; defined \; as \; P(0, x_1, \cdots, x_n) \wedge P(1, x_1, \cdots, x_n) \wedge \cdots \wedge P(y, x_1, \cdots, x_n) \wedge \cdots$$

Definition 4. *An n-ary predicate is a well-formed formula (wff) of first-order logic. Any expression formed out of first-order logic wffs is also a wff. Nothing else is a wff.*

Definition 5. *An interpretation M consists of an non-empty set D, called the domain of the interpretation, and an assignment to each predicate letter A_j^n of a function from D^n to the set comprised of two values true and false (a set of finite sequences of elements in D), to each n-ary function constant f_j^n an n-ary function from D^n to D, and to each individual constant a_i of some fixed element $(a_i)^M$ of D. A valuation is the assignment of an element of the domain D to each variable. For a given interpretation, the truth table of any formula is defined by the following rules:*

- *The truth tables for propositional connectives apply to evaluate the value of (F AND G), (F OR G), (F implies G), and (NOT F).*
- *("for all x, F") is true if F is true for every element x of D. Otherwise, is false.*
- *("there exists an x such that F") is true if F is true for at least one element x of D. Otherwise, is false.*

Definition 6. *An interpretation satisfies a wff if the wff has the value true under this interpretation. An interpretation that satisfies a set of wffs is a model for this set.*

Definition 7. *A finite automaton M on the alphabet $A = s_1, \cdots, s_n$ with states $Q = q_1, \cdots, q_m$, is given by a function δ which maps each pair $(q_i, s_j), 1 \leq i \leq m, 1 \leq j \leq n$, onto a state q_k, together with a set $F \subseteq Q$, where F is the set of accepting states. State q_1 is called the initial state.*

Definition 8. *An alphabet is a nonempty set A of objects called symbols. An n-tuple of symbols of A is called a word or a string on A. For simplicity, a word u is written as $u = a_1 a_2 \cdots a_n$. n is the length of $u, |u| = n$. A unique null word is allowed, 0. The set of all words in the alphabet A is written A^*. Any subset of A^* is called a language on A or a language with alphabet A. A word of length 1 which contains a symbol a_i is the same as the symbol itself.*

Let M be a finite automaton with transition function δ, initial state q_1 and accepting states F. If q_i is any state of M and $u \in A^*$, where A is the alphabet of M, the $\delta^*(q_i, u)$ is the state which M will enter if it begins in state q_i at the left end of the string u and moves across u until the entire string has been processed. M accepts a word u provided that $\delta^*(q_1, u) \in F$. M rejects u means that $\delta^*(q_i, u) \in Q - F$. Finally, the language accepted by M, written $L(M)$ is the set of all $u \in A^*$ accepted by M:

$$L(M) = \{u \in A^* | \delta^*(q_1, u) \in F\}.$$

Definition 9. *A language is called regular if there exists an automaton that accepts it.*

It is often very useful to represent the transition function δ graphically. Given a graph where each state is represented with a *vertex*, then the fact that $\delta(q_i, s_j) = q_k$ is represented by drawing an arrow from vertex q_i to vertex q_k and labelling it s_j. The diagram thus obtained is called the *state transition diagram* for the given automaton.

A.2 Limitation of FSMs in Reasoning with Negation as Lack of Information

We assume a predicate $X = H\_UserInLocation(x, rid)$, $x \in Users, rid \in Locations$ [1]. We assume that the set of Users consists of symbols $\{a_1, a_2\}$ and the set of Locations consists of symbols $\{r_1, r_2\}$. We assume that for each event of type $H\_UserInLocation$, an instance of the *H_UserInLocation* predicate (a state) is created as a symbol that can be passed on to a finite state machine. We refer to this symbol as $UL(x, rid)$.

[1] For reasons of simplicity the *role* and *rattr* are omitted from the predicate.

User a_1 is Nowhere.

The expression:

$$\neg H\_UserInLocation(a_1, y) \tag{A.1}$$

is equivalent to *"User a_1 is nowhere"*. This is implemented with the FSM M_1 of Figure A.1.

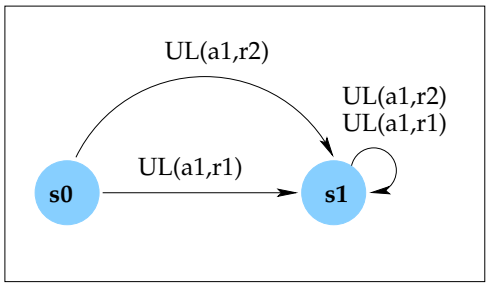


Figure A.1. $\neg H\_UserInLocation(a_1, y)$

The alphabet for M_1 is the set $A = \{UL(a_1, r_1), UL(a_1, r_2), UL(a_2, r_1), UL(a_2, r_2)\}$. The set of states for M_1 is $Q = \{s_0, s_1\}$. The set of accepting states F is the empty set. The function δ is defined as follows: State s_0 is the initial state of the FSM. When either the symbol $UL(a_1, r_1)$ or $UL(a_1, r_2)$ is passed to state s_0, the FSM makes a transition from state s_0 to state s_1, which is a non-accepting state for this FSM. From s_1, when either of the symbols $UL(a_1, r_1)$,$UL(a_1, r_2)$ are encountered, M_1 remains in state s_1.

As can be easily seen from the above, M_1 does not have any accepting states. State s_1 is a non-accepting state that corresponds to the inverse statement to the one of interest, namely that *user a_1 is somewhere*. State s_0 is also a non-accepting state, where the FSM *doesn't know yet* whether user a_1 is anywhere or not. Therefore, M_1 does not accept any language, including Expression A.1.

It can be assumed from this example that FSMs are *not sufficient* in dealing with *lack of information*. Note that the same problem arises with any FOL operator that signifies *lack of information* such as $\nexists$, as is illustrated in the next example.

There Doesn't Exist any Meeting Room On-Site

The FOL expression:

$$\nexists x\,(AtomicLocation(x, \textit{Meeting Room})) \tag{A.2}$$

is also not directly implementable by a FSM for lack of an accepting state.

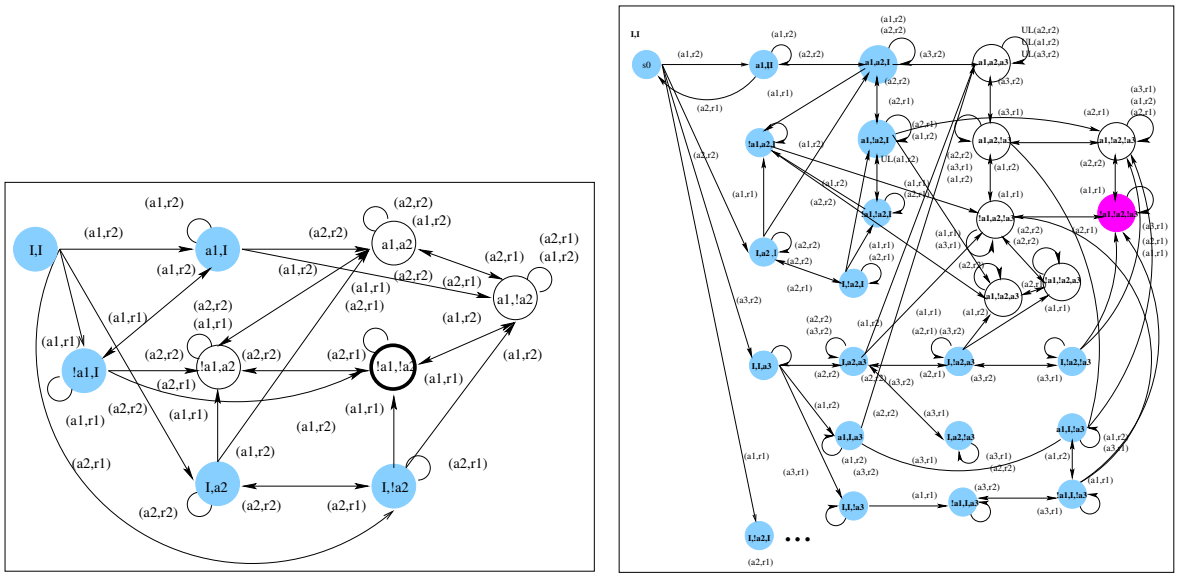
A.3 The Closed World Assumption

Assuming a closed world that consists of two regions r_1 and r_2 and users a_1 and a_2:

$$\nexists x\,(H\_UserInLocation(x, r_2)) \tag{A.3}$$

Equation A.3 signifies that *there is nobody in room r_2* or in other words, that *room r_2 is empty*. However, this is equivalent to the expression *everybody is in room r_1*:

$$H\_UserInLocation(x, r_1) \tag{A.4}$$



(a) (b)

Figure A.2. FSM implementations for domains D_1 (a) and D_2 (b) for Expression A.3

An attempt to model the above expression with the FSM M_2, which is portrayed in Figure A.2(a) is discussed here. M_2 accepts the alphabet $A = \{UL(a_1, r_1), UL(a_1, r_2), UL(a_2, r_1), UL(a_2, r_2)\}$. States in magenta represent FSM accepting states. States in blue represent *partial knowledge* of whether the expression is satisfied or not. This is particularly obvious in the case where the FSM is restarted. Until all of the symbols $\{UL(a_1, y), UL(a_2, x), \forall x, y \in Locations\}$ have passed through the FSM, there is no way of knowing whether the expression is true or not. The letter I is used to denote *ignorance* of the state of the respective symbol. For example, state s_0 is also symbolised as I, I to denote that initially no knowledge is held for either of users a_1, a_2.

In fact, quite often in sensor-driven networks, if a user is idle, notification about this user will be retained until the user starts moving again. If, when the FSM is restarted, user a_1 is idle in region r_2, the FSM will be in a state of ignorance for that user until the user moves again.

A.4 Technical Background on FSMs with Free Variables

This section illustrates the shortcomings of current implementations, based on FSMs without constants, for evaluating FOL expressions of state.

FSMs that implement expressions without constants are referred to as *parametric FSMs*, and they have been presented in [5, 39]. Such FSMs evaluate event-histories without constants. With the exception of [5], these implementations have been used to evaluate event histories which "happen" and cannot be undone, rather than states that can be activated or de-activated as a result of event occurrences. This means that for reasoning with state, each model state that is modelled by an FSM state needs to be checked for whether it still holds, or not with each event occurrence. When the state includes a free

variable, transitions to the negation of the free variable need to be designed. Assuming a closed world, such transitions are implicit disjunctions over the rest of the mutually exclusive alternatives (except the symbol that has already occurred) and, as a result, there is no automatic way to create these transitions with the existing methods, as there is no way of predicting in advance what the value of the free variable will be. This is illustrated in the following examples that use the two methods from the literature that deal with FSMs with free variables. We present briefly both methods and illustrate their shortcomings for the case of state representation of sensor-driven systems by presenting counterexamples that can't be implemented directly with either of the two methods.

Spawning Parametric FSMs

We refer to a methodology presented in [39] as the *spawning parametric FSMs* method. This method involves spawning, for each free variable in the initial parametric FSM, an identical non-parametric FSM whenever a symbol that instantiates a parameter occurs at a given state. In the spawned FSM, all instances of the parameter that corresponds to the symbol that has been encountered are substituted with the actual symbol in all transitions of the FSM. Similarly to simple FSMs, a parametric FSM in this method is said to accept an expression that contains free variables, whenever one of the spawned FSMs reaches an accepting state.

Multi-Bead Parametric FSMs

The second method is proposed in [5], and it constitutes an enhancement of the spawning parametric FSM method by allowing concurrent processing of alphabet symbols. This is achieved by using symbols called *beads*. Each time an alphabet symbol needs to be processed, the corresponding *bead* is created. As a bead traverses the states, it records the values of all events which cause it to move in its *path*. Every time a symbol arrives at a state that has a transition where a free variable represents that bead, a new FSM is spawned where a constant is assigned to a variable. Concurrent processing is implemented by having more than one bead in the FSM at one instant. A parametric-FSM in this method is said to accept the expression that is logged in the bead's path when a bead reaches an accepting state.

A.4.1 Counterexample 1

Consider the following parametric expression (Expression A.5):

$$(H\_UserInLocation(x_1, y) \wedge H\_UserInLocation(x_2, y))$$

This expression signifies that *Users* x_1 *and* x_2 *are co-located.*

Implementing Expression A.5 with Spawning Parametric FSMs.

Figure A.3 illustrates this with an example. In Figure A.3(a), if an event occurs that corresponds to user a_1 being inside region r_1, then the automaton of Figure A.3(b) will be spawned from the one in (a). In this automaton, the transitions from s_1 to s_3 must represent all events that report user a_1 exiting from region r_1. This means that there must exist one transition for each region in the universe which is different from r_1. If instead user a_1 had moved into r_2, the automaton that would need to be constructed as a result would be that of Figure A.3(c).

Implementing Expression A.5 with Multi-Bead Parametric FSMs

.

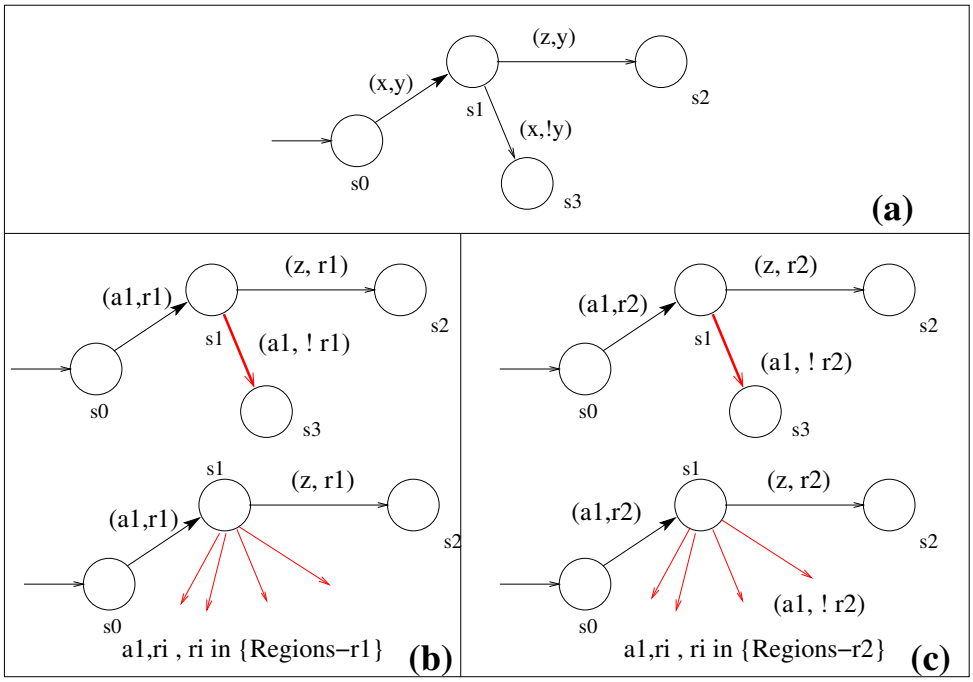


Figure A.3. $H\_UserInLocation(x, y) \wedge H\_UserInLocation(z, y)$(non-concurrent processing approach)

A detail of a finite state machine with spawning arcs similar to the ones described in [5] is shown in Figure A.4. At state s_1, if it is detected that user A leaves room x, its bead will move to state s_2 where it will be destroyed. The same problem arises as in the previous case, namely, that according to the value of x of the event that makes the automaton to change from state s_0 to state s_1, a transition for each value $y \in \{Locations - x\}$ needs to be created to state s_2. Another shortcoming of this method is that whenever a confirmatory event of a user's position is received, a new bead for this position will be created from state s_0 to state s_1, and this is clearly wrong.

A.4.2 Counterexample 2

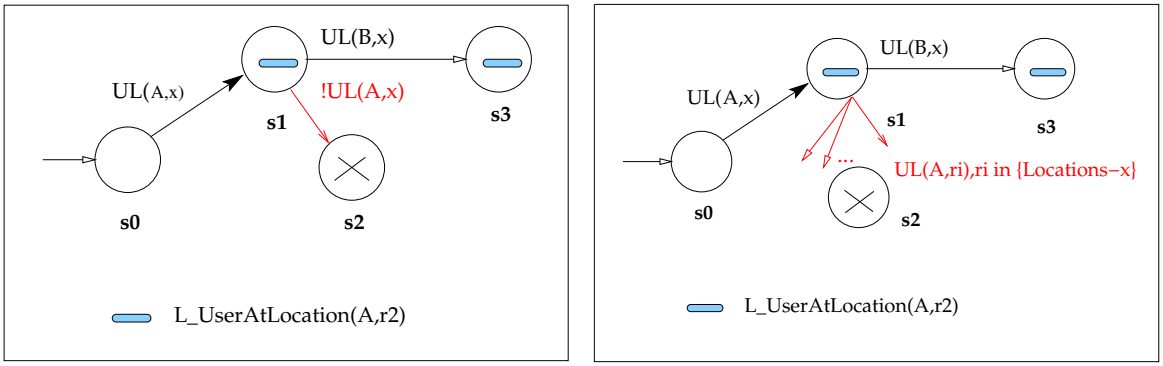
This counter example considers higher-level state predicates. The following set of FOL expressions signify a situation where two users x_1, x_2 are co-located while x_2 is walking and a third user x_3 is sitting down.

$$
\begin{aligned}
&H\_UserInLocation(x_1, y) \wedge H\_UserInLocation(x_2, y) \Rightarrow UsersAreColocated(x_1, x_2, y)\\
&UsersAreColocated(x_1, x_2, y) \wedge UserMovement(x_3, Sitting) \Rightarrow\\
&UsersAreColocatedWhileMovement(x_1, x_2, y, x_3, Sitting)\\
&UsersAreColocatedWhileMovement(x_1, x_2, y, x_3, Sitting) \wedge UserMovement(x_2, Walking) \Rightarrow\\
&UsersAreColocatedWhileMovement2(x_1, x_2, y, x_3, Sitting, x_2, Walking)
\end{aligned}
\tag{A.5}
$$

A detail of an FSM for the above expression is portrayed in Figure A.5(a). Because the predicate

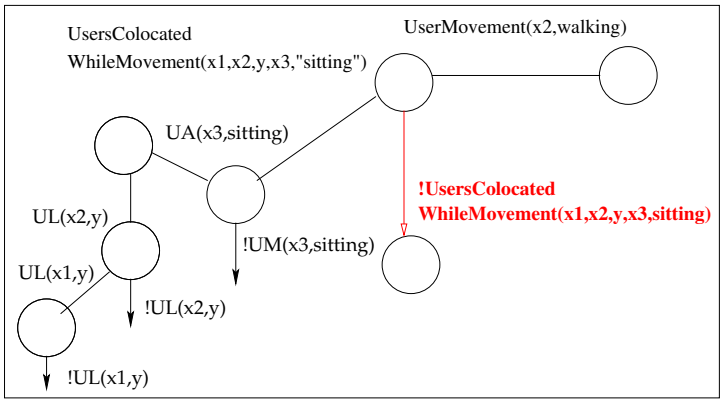
$$\neg UsersAreColocatedWhileMovement(x_1, x_2, y, x_3, Sitting)$$

is not by default available in the system, the only way to determine when the transition with this label is

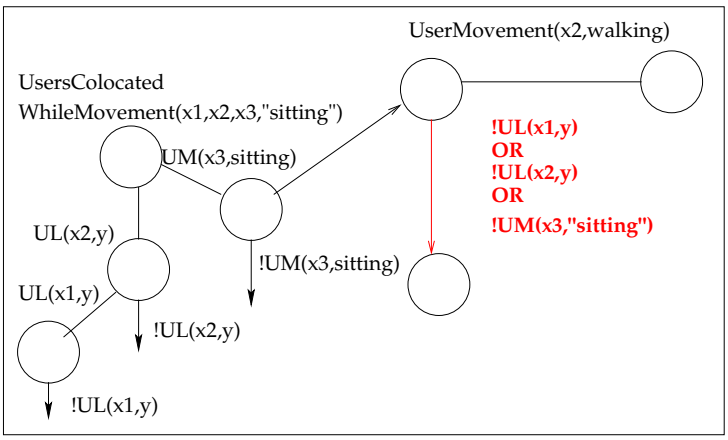


(a) (b)

Figure A.4. $H\_UserInLocation(x, y) \wedge H\_UserInLocation(z, y)$(multi-bead method)



(a)



(b)

Figure A.5. x_1, x_2 are co-located while x_2 is walking and x_3 is sitting down.

satisfied is to determine whether the equivalent expression

$$\neg H\_UserInLocation(x_1, y) \vee \neg H\_UserInLocation(x_2, y) \vee \neg UserMovement(x_3, Sitting)$$

is satisfied, as shown in Figure A.5(b). As in the general case, the predicates $\neg H\_UserInLocation(x_1, y)$, $\neg H\_UserInLocation(x_2, y)$ and $\neg UserMovement(x_3, Sitting)$ are not available in the system unless they have been defined through a binding definition; in order to decide whether they are satisfied or not, the equivalent expressions involving primitive state predicates need to be evaluated (using the closed world assumption) which makes the implementation domain-specific. This violates CAAT.

As can be concluded from the above example, the absence of the negative predicates in the model and the incapability of finite state machines to deal with lack of information do not allow the use of FSMs for detecting state in a transparent way.

Appendix B

SCALA Modules

This section summarises the SCALA modules (libraries) and the contained application programmable interfaces by discussing, wherever appropriate, the architecture of the software component that corresponds to each module.

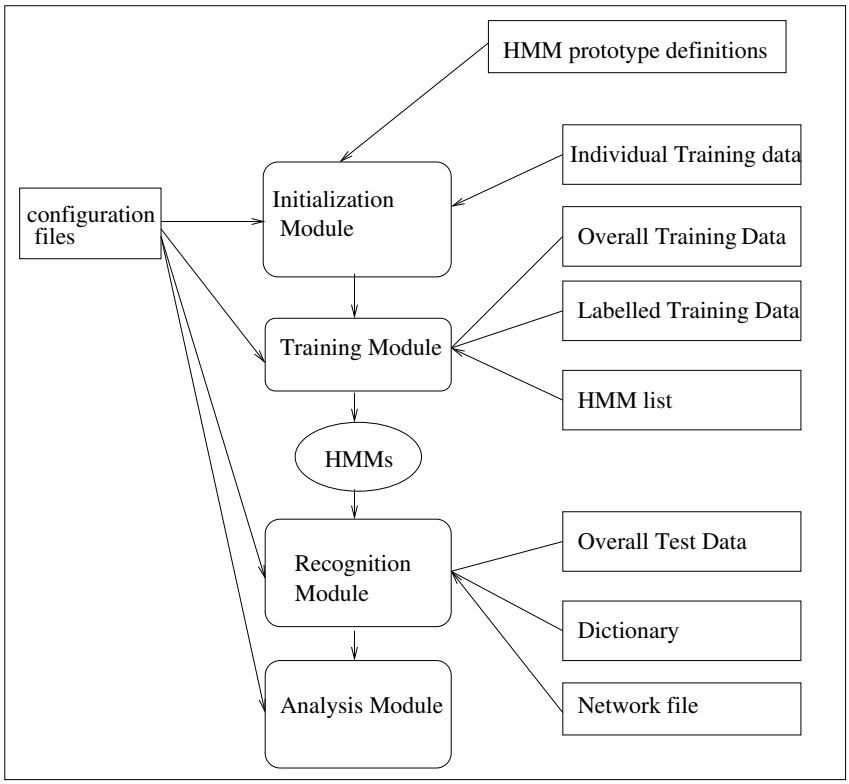
B.0.3 The Deductive Knowledge Base Module

The architecture of the Deductive KB component is discussed in Chapter 10 (Figure 10.3). Based on the above architecture, SCALA supports the following statements for building and using the Deductive KB component.

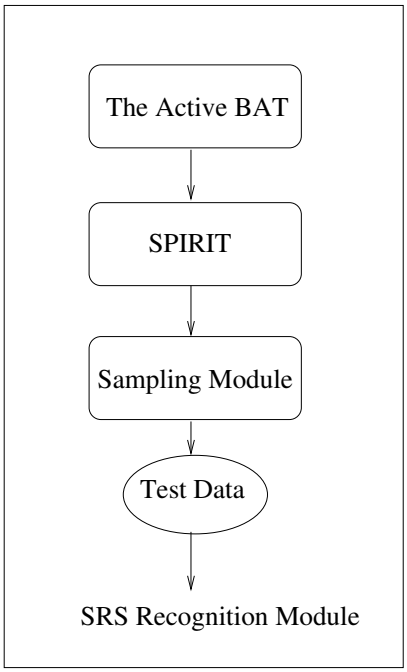
- CreateReteManager()
- CreateEventAdaptor()
- CreateEventListener()
- CreateSPIRITEventListener()
- CreateKnowledgeBase()
- GetKnowledgeBase()
- UserUpdate()
- SensorUpdate()
- UserQuery()
- ExportFOLModel()
- InitKnowledgeBase()
- LoadConfiguration()
- Reset()
- Clear()

B.0.4 The SCALA Statistical Inference Service Module

The Statistical Inference Service module is based on the methodology discussed in Chapters 3 and 4. It consists of the Statistical Recognition Service (SRS) module and the Probability Estimation Service (PES) module. The SRS module implements the methodology described in Chapter 3 and the PES module the methodology described in Chapter 4.



(a) SRS



(b) The Sampling module

Figure B.1. The Statistical Inference Service module

Statistical Recognition Service API

The architecture of the SRS is portrayed in Figure B.1(a) and it consists of a *sampling module*, an *initialisation module*, a *training module*, a *recognition module* and an *analysis* (evaluation) *model*. The sampling module (Figure B.0.4) samples streams of position events and produces observation vectors that are tested using the recognition module. The initialisation and training modules take as input a set of files (configuration, model prototype definition and training data) and iteratively computes a set of models that represent the phonemes of Chapter 3.

The recognition module takes as input a network and a dictionary file describing the allowable phoneme sequences, a set of HMM models and a set of test data. It then performs recognition of the test data by selecting the HMM model with the highest probability.

Once the recogniser is built, it is necessary to evaluate its performance. This is done by comparing the output with a set of correct data. This comparison is performed by the analysis module.

SCALA provides support for the online creation of the SRS, as well as for its online usage with location data (see also Figure 10.1). The CreateSRS-KSLink() statement implements the interface to the Deductive KB component.

- CreateSamplingModule()
- CreateSRSInitializationModule()
- CreateSRSTrainingModule()
- CreateSRSAnalysisModule()
- DoInitialization()
- DoTraining()
- DoRecognition()
- DoAnalysis()
- AddObservationVector()
- AddObservationList()
- AddConfigurationParameters()
- AddHMMList()
- AddNetworkFile()
- AddDictionary()
- MovementUpdate()
- SensorUpdate()
- CreateSRS-KSLink()

The SCALA Probability Estimation Service Module

The Probability Estimation Service (PES) implements the methodology of Chapter 4. It takes as input specifically formatted historical data, a set of arguments that define the model's operation and the predicate for which the probability is to be estimated, and it produces a probability estimate. The CreatePES-KSLink() statement implements the interface to the Deductive KB component.

SCALA provides the following APIs for creating and using PES.

- CreatePESModule()
- AddPESParameterFile()
- AddDataFile()

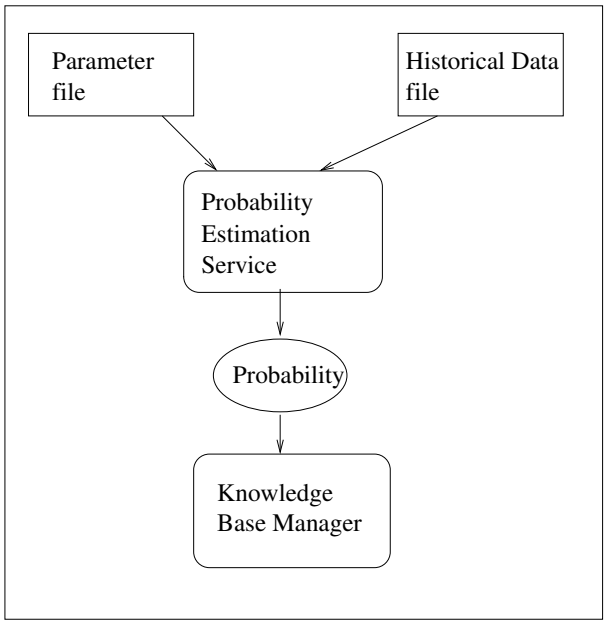


Figure B.2. The Probability Estimation Service architecture

- Fact()
- FactLikelihoodUpdate()
- CreatePES-KBLink()

B.0.5 The SCALA Satisfiability Service Module

The Satisfiability Service determines whether the user requirements are satisfiable based on the knowledge that is maintained in the knowledge base (see Figure 9.1). SCALA supports the Satisfiability Service with the following statements.

- CreateSatisfiabilityService()
- CheckSatisfiable()
- LoadFOLModel()
- AddFOLFormula()

B.0.6 The SCALA AED Service Module

The Abstract Event Detection Service is a higher order service for abstract events in sensor-driven systems described in Chapter 8. SCALA supports the AED Service by providing a set of APIs for the creation and control of this service. More specifically:

- CreateAEDService()
- Publish()
- Subscribe()

B.0.7 The SCALA Context-Aware Application Module

The ECA Service is described in Chapter 12. SCALA provides a set of APIs that support the creation and control of the ECA Service, namely:

- CreateECAApp()
- RemoveECAAPP()
- TriggerAction()

B.0.8 The SCALA SPIRIT Module

The SCALA language provides support for using the SPIRIT system as a source for SPIRIT events. It also offers object interfaces, i.e., it allows the programmer to manipulate the SPIRIT database objects.

B.0.9 SCALA Support for the SPIRIT Module

- CreateSPIRITListener()
- GetSPIRITObject()
- CreateSPIRITEventsConsumer()
- CreateSPIRIT-KBLink()

B.0.10 The SCALA QoSDREAM Module

The SCALA Language provides support for connecting to the QosDREAM system mainly as an event source but also offering object interfaces.

- The SCALA support for the QoSDREAM module
- CreateQoSDREAMListener()
- GetQoSDREAMObject()
- CreateQoSDREAMEventsConsumer()
- CreateQoSDREAM-KSLink()

B.0.11 The SCALA Generic Module

This module provides a wrapper over a generic context source which causes it to publish an event interface. This is then linked into the Deductive Knowledge Base component by means of a CORBA event manager. This is used for legacy systems that can in this way be integrated into the system as context-sources.

- CreateKBLink()

Bibliography

[1] N. Adly, P. Steggles, and A. Harter. SPIRIT: A Resource Database for Mobile Users. In *CHI '97: Proceedings of Conference on Human Factors in Computing Systems*, Atlanta, GA, Mar. 1997. ACM Press.

[2] Aristotle. De Anima. http://psychclassics.yorku.ca/Aristotle/De-anima/de-anima1.htm.

[3] Aristotle. Posterior Analytics. http://classics.mit.edu/Aristotle/posterior.html.

[4] D. Ashbrook and T. Starner. Learning Significant Locations and Predicting User Movements with GPS. In *Proceedings of the 6th International Symposium on Wearable Computers*, pages 7–10, Seattle, WA, Oct. 2002. IEEE Computer Society Press.

[5] J. Bacon, J. Bates, R. Hayton, and K. Moody. Using Events to Build Distributed Applications. In *Proceedings of the 2nd International Workshop on Services in Distributed and Network Environments*, pages 148–155, Whistler, Canada, June 1995. IEEE Computer Society Press.

[6] J. Bacon and K. Moody. Toward Open, Secure, Widely Distributed Services. *Communications of the ACM*, 45(6):59–64, 2002.

[7] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri. Generic Support for Distributed Applications. *IEEE Computer*, 33(3):68–76, 2000.

[8] J. Bacon, K. Moody, and W. Yao. A Model of OASIS Role-Based Access Control and its Support for Active Security. *ACM Transactions on Information Systems Security*, 5(4):492–540, 2002.

[9] P. Bahl and V.N. Padmanabhan. RADAR: An In-Building RF-Based User Location and Tracking System. In *Proceedings of IEEE INFOCOM 2000 (2)*, pages 775–784, Tel-Aviv, Israel, Mar. 2000. IEEE Computer Society Press.

[10] A. Belokosztolszki, D. Eyers, P. Pietzuch, J. Bacon, and K. Moody. Role-Based Access Control for Publish/Subscribe Middleware Architectures. In *DEBS '03: Proceedings of the 2nd International Workshop on Distributed Event-Based Systems*, pages 1–8, San Diego, CA, June 2003. ACM Press.

[11] A. R. Beresford and F. Stajano. Location Privacy in Pervasive Computing. *IEEE Pervasive Computing*, 2(1):46–55, Jan.–Mar. 2003.

[12] P. Bonnet, J. Gehkre, and P. Seshadri. Towards Sensor Database Systems. In *Proceedings of the 2nd International Conference on Mobile Data Management*, pages 3–14, Hong Kong, China, Jan. 2001. Springer-Verlag.

[13] P. J. Brown. The Stick-e Document: A Framework for Creating Context-Aware applications. In *Proceedings of EP'96*, pages 259–272, Palo Alto, CA, Jan. 1996. John Wiley and Sons.

[14] L. Cardelli and A. Gordon. Mobile Ambients. In *FoSSaCS '98: Proceedings of the 1st International Conference on Foundations of Software Science and Computation Structures*, pages 140–155, Lisbon, Portugal, Mar.–Apr. 1998. Springer-Verlag.

[15] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[16] The Castor Project. http://www.castor.org.

[17] D. Caswell and P. Debaty. Creating Web Representations for Places. In *HUC '00: Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing*, pages 114–126, Bristol, UK, Sep. 2000. Springer-Verlag.

[18] E. M. Clark, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[19] CLIPS. A Tool for Designing Expert Systems. http://www.ghg.net/clips/CLIPS.html.

[20] CLIPS. User's Guide version 6.10. http://herzberg.ca.sandia.gov/jess/docs/index.shtm.

[21] CORBA. http://www.CORBA.org.

[22] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems (3rd ed.): Concepts and Design*. Addison-Wesley, 2001.

[23] G. Coulouris, H. Naguib, and S. Mitchell. Middleware Support for Context Aware Multimedia Applications. In *DAIS '01: Proceedings of the 3rd IFIP International Working Conference on Distributed Applications and Interoperable Systems*, pages 9–22, Krakov, Poland, Sep. 2001. Kluwer.

[24] M. E. Crovella and T. J. LeBlanc. Performance Debugging Using Parallel Performance Predicates. In *PADD '93: Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 140 – 150, San Diego, CA, May 1993. ACM Press.

[25] J. Crowcroft, J. Bacon, P. Pietzuch, G. Coulouris, and H. Naguib. Channel Islands In A Reflective Ocean: Large Scale Event Distribution in Heterogeneous Networks. *IEEE Communications Magazine*, 40(9):112–115, Sep. 2002.

[26] P. Dana. Global Posioning System Overview. http://www.colorado.edu/geography/gcraft/notes/gps/gps.htm.

[27] C. J. Date and H. Darwen. *A Guide to the SQL Standard (4th ed.): A User's Guide to the Standard Database Language SQL*. Addison-Wesley, 1997.

[28] R. W. DeVaul, B. Clarkson, and A. S. Pentland. The Memory Glasses, Towards a Wearable Context-Aware, Situation-Appropriate Reminder System. In *Proceedings of Workshop on Situated Interaction in Ubiquitous Computing at CHI '00*, The Hague, The Netherlands, Apr. 2000.

[29] A. K. Dey. Understanding and Using Context. *Personal and Ubiquitous Computing*, 5(1):4–7, 2001.

[30] A. K. Dey and G. D. Abowd. CyberMinder: A Context-Aware System for Supporting Reminders. In *HUC '00: Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing*, pages 172–186, Bristol, UK, Sep. 2000. Springer-Verlag.

[31] A. K. Dey and G. D. Abowd. *Providing Architectural Support for Building Context-Aware Applications*. PhD thesis, Georgia Institute of Technology, 2000.

[32] R. Dugad. A Tutorial on Hidden Markov Models. Technical Report SPANN-96.1, Indian Institute of Technology, 1996.

[33] P. Th. Eugster, P. A. Felber, R. Guerraoui, and A. M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[34] R. Fleming and C. Kushner. Low-Power, Miniature, Distributed Position Location and Communication Devices Using Ultra-Wideband, Non-Sinusoidal Communication Technology. Technical report, Aether Wire Location, 1995.

[35] A. Flew and S. Priest. *A Dictionary of Philosophy*. Pan Books, 2002.

[36] Laboratory for Communication Engineering (LCE). http://www-lce.eng.cam.ac.uk.

[37] C. L. Forgy. *On the Efficient Implementation of Production Systems.* PhD thesis, Carnegie-Mellon University, 1979.

[38] C. L. Forgy. Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern-Match Problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[39] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model and Implementation. In *Proceedings of the 18th International Conference on Very Large Databases*, pages 327–338, Vancouver, Canada, Aug. 1992. Morgan Kaufmann.

[40] E. N. Hanson, S. Bodagala, M. Hasan, G. Kulkarni, and J. Rangarajan. Optimized Rule Condition Testing in Ariel Using Gator Networks. Technical Report TR-95-027, CISE Department, University of Florida, Gainesville, FL, 1995.

[41] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The Anatomy of a Context-Aware Application. In *MobiCom '99: Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 59–68, Seattle, WA, Aug. 1999. ACM Press.

[42] Robert Headon. Movement Awareness for a Sentient Environment. In *PerCom '03: Proceedings of the 1st Conference on Pervasive Computing and Communications*, pages 99–106, Fort Worth, TX, Mar. 2003. IEEE Computer Society Press.

[43] J. H. Hine, W. Yao, J. Bacon, and K. Moody. An Architecture for Distributed OASIS Services. In *Middleware '00: IFIP/ACM International Conference on Distributed systems platforms*, pages 104–120, New York, NY, 2000. Springer-Verlag.

[44] A. Hopper. The Clifford Paterson Lecture: Sentient Computing. *Philosophical Transactions of the Royal Society of London*, 358(1773):2349–2358, August 1999.

[45] E. Horvitz. Principles of Mixed-Initiative User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 159–166, Pittsburgh, PA, May 1999. ACM Press.

[46] D. Ipiña and E. Katsiri. A Rule-Matching Service for Simpler Develpment of Reactive Applications. *IEEE Distributed Systems Online*, 2(7), 2001.

[47] The Reference Model of Open Distributed Processing, ITU-T Recommendation X901— ISO/IEC 10746-1:Overview, 1998.

[48] The Reference Model of Open Distributed Processing, ITU-T Recommendation X901— ISO/IEC 10746-2:Foundations.

[49] The Reference Model of Open Distributed Processing, ITU-T Recommendation X901— ISO/IEC 10746-3:Architecture.

[50] The Java Beans Technology. http://java.sun.com/products/javabeans/.

[51] Jess: The Rule Engine for the Java Platform. http://herzberg.ca.sandia.gov/jess.

[52] E. Katsiri. Principles of Context Inferences. In *Adjunct Proceedings of Ubicomp '02*, pages 33–34, Gothenburg, Sweden, Oct. 2002.

[53] E. Katsiri and A. Mycroft. Knowledge-Representation and Abstract Reasoning for Sentient Computing. In *Proceedings of 1st Workshop on Challenges and Novel Applications of Automated Reasoning, in conjunction with CADE-19*, pages 73–82, Miami Beach, FL, Jul.–Aug. 2003.

[54] G. Kortuem, Z. Segall, and Th. G. Cowan. Close Encounters: Supporting Mobile Collaboration Through Interchange of User Profiles. *Lecture Notes in Computer Science*, 1707:171–185, 1999.

[55] J. Krumm, L. Williams, and G. Smith. SmartMoveX on a Graph - An Inexpensive Active Badge Tracker. In *Ubicomp '02: Proceedings of the 4th International Conference on Ubiquitous Computing*, pages 299–307, Gothenburg, Sweden, Sep. 2002. Springer-Verlag.

[56] T. Kunz. An Event Abstraction Tool: Theory, Abstraction and Results. Technical report, Technical University Darmstat, 1994.

[57] T. Kunz. Visualizing Abstract Events. In *CASCON '94: Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 334–343, Toronto, Canada, Nov. 1994. IBM Press.

[58] C. Liebig, M. Cilia, and A. Buchmann. Event Composition in Time-Dependent Distributed Systems. In *CoopIS '99: Proceedings of the 4th IFCIS International Conference on Cooperative Information Systems*, pages 70–78, Edinburgh, UK, Sep. 1999. IEEE Computer Society Press.

[59] D. Lopez de Ipina. TRIP: A Distributed Vision-Based Sensor System. Technical report, University of Cambridge, 1999.

[60] S. Madden and M. Franklin. Fjording the Stream: an Architecture for Queries over Streaming Data. In *Proceedings of the ICDE Conference*, pages 555–566, San Hose, CA, Feb.–Mar. 2002. IEEE Computer Society Press.

[61] Object Management Group (OMG). Notification Service Specification, 2000.

[62] S. Mann. Wearable Computing as means for Personal Empowerment. Keynote Address for the ICWC-98: 1st International Conference on Wearable Computing, May 1998.

[63] M. Mansouri-Samani and M. Sloman. GEM: A Generalised Event Monitoring Language for Distributed Systems. *Distributed Systems Engineering Journal*, 4(2):96–108, June 1997.

[64] N. Marmasse. comMotion: A Context-Aware Communication System. In *CHI '99: Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 320–321, Pittsburg, PA, May 1999. ACM Press.

[65] E. Mendelson. *Introduction to Mathematical Logic (3rd ed.)*. Wadsworth, 1987.

[66] S. Mitchell, M. D. Spiteri, J. Bates, and G. Coulouris. Context-Aware Multimedia Computing in the Intelligent Hospital. In *EW 9: Proceedings of the 9th Workshop on ACM SIGOPS European Workshop*, pages 13–18, Kolding, Denmark, Sep. 2000. ACM Press.

[67] T. M. Mitchell. *Machine Learning*. Mc Graw-Hill, 1997.

[68] A. L. Murphy, G. P. Picco, and G.-C. Roman. Developing Mobile Computing Applications with Lime. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, pages 766–769, Limerick, Ireland, June 2002. ACM Press.

[69] P. Myllymaki, T. Silander, H. Tirri, and P. Uronen. B-Course: A Web-Based Tool for Bayesian and Causal Data Analysis. *International Journal on Artificial Intelligence Tools*, 11(3):369–387, 2002.

[70] MySQL: The World's Most Popular Open Source Database. http://www.mysql.com.

[71] H. Naguib and G. Coulouris. Location Information Management. In *UBICOMP '01: Proceedings of the 3rd International Conference on Ubiquitous Computing*, pages 35–41. Springer-Verlag, Sep.–Oct. 2001.

[72] A. K. Narayanan. Realms and States: A Framework for Location-Aware Mobile Computing. In *WMC '01: Proceedings of the 1st International Workshop on Mobile Commerce (MobiCom)*, pages 48–54, Rome, Italy, July 2001. ACM Press.

[73] D. Nardi and R. J. Brachman. *The Description Logic Handbook*, chapter 1:An Introduction to Description Logics, pages 5–44. Cambridge University Press, 2002.

[74] N. J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, 1998.

[75] A. Nonnengart, G. Rock, and C. Weidenbach. On Generating Small Clause Normal Forms. In *CADE-15: Proceedings of the 15th International Conference on Automated Deduction*, pages 397–411, Lindau, Germany, July 1998. Springer-Verlag.

[76] B. O'Conaill and D. Frohlich. Timespace in the Workplace: Dealing with Interruptions. In *CHI '95: Conference Companion on Human Factors in Computing Systems*, pages 262–263, Denver, CO, May 1995. ACM Press.

[77] OSI 7498: Open System Interconnection Seven-Layer Model. http://www.acm.org/sigcomm/standards/iso_stds/OSI_MODEL.

[78] P. Pietzuch and J. Bacon. HERMES: A Distributed Event-Based Middleware Architecture. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 611–618, Vienna, Austria, July 2002. IEEE Computer Society Press.

[79] P. Pietzuch and J. Bacon. HERMES: A Distributed Event-Based Middleware Architecture. *IEEE Distributed Systems Online*, 2002.

[80] P. Pietzuch and J. Bacon. Peer-to-Peer Overlay Broker Networks in an Event-Based Middleware. In *DEBS '03: Proceedings of the 2nd International Workshop on Distributed Event-Based Systems*, pages 1–8, San Diego, CA, July 2003. ACM Press.

[81] P. Pietzuch, B. Shand, and J. Bacon. Composite Event Detection as a Generic Middleware Extension. *IEEE Network Magazine, Special Issue on Middleware Technologies for Future Communication Networks*, Jan.-Feb. 2004.

[82] Plato. Cratylus. http://classics.mit.edu/Plato/cratylus.html.

[83] Plato. Meno. http://classics.mit.edu/Plato/meno.htm.

[84] Plato. Republic. http://classics.mit.edu/Plato/republic.html.

[85] Plato. Theaetetus. http://classics.mit.edu/Plato/meno.html.

[86] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The Cricket Location-Support System. In *MobiCom '00: Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, pages 32–43, Boston, MA, Aug. 2000. ACM Press.

[87] QoSDREAM: Quality of Service for Distributed REconfigurable Adaptive Multimedia. http://www-lce.eng.cam.ac.uk/qosdream/.

[88] L. R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[89] K. Rehman. 101 Ubiquitous Computing Applications. http://www-lce.eng.cam.ac.uk/ kr241/html/101_ubicomp.html, 2001.

[90] B. J. Rhodes. The Wearable Remembrance Agent: a System for Augmented Memory. In *ISWC '97: Proceedings of the 1st IEEE International Symposium on Wearable Computers*, pages 123–128, Cambridge, MA, Oct. 1997. IEEE Computer Society Press.

[91] Tristan Richardson. Teleporting: Mobile X Sessions. *The X Resource*, 13(1):133–140, 1995.

[92] A. Rowstron and P. Druschel. PASTRY: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware '01: Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329–350, Heidelberg, Germany, Nov. 2001. Springer-Verlag.

[93] A. Rowstron, A. Kermarrec, and M. Castro. SCRIBE: The Design of a Large-Scale Event Notification Infrastructure. In *NGC 2001: Proceedings of the 3rd International Workshop on Networked Group Communications*, pages 30–43, London, UK, Nov. 2001. Springer-Verlag.

[94] D. Salber, A. K. Dey, and G. D. Abowd. The Context-Toolkit: Aiding the Development of Context-Enabled Applications. In *CHI '99: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 434–441. ACM Press, 1999.

[95] N. Sawhney and C. Schmandt. Nomadic Radio: Speech and Audio Interaction for Contextual Messaging in Nomadic Environments. *ACM Transactions on Computer-Human Interaction*, 7(3):353–383, 2000.

[96] B. Schilit, N. Adams, and R. Want. Context-Aware Computing Applications. In *Proceedings of Workshop on Mobile Computing Systems and Applications*, Santa-Cruz, CA, Dec. 1994. IEEE Computer Society.

[97] B. N. Schilit, N. Adams, R. Gold, M. Tso, and R. Want. The PARCTAB Mobile Computing System. Technical Report CSL-93-20, Xerox PARC, Oct. 1993.

[98] A. Schmidt, M. Beigl, and H.W. Gellersen. There is More to Context than Location. *Computers and Graphics*, 23(6):893–901, 1999.

[99] D. Scott, A. Beresford, and A. Mycroft. Spatial Security Policies for Mobile Agents in a Sentient Computing Environment. In *Proceedings of the IEEE 4th International Workshop for Policies for Distributed Systems and Networks*, pages 147–157, Lake Como, Italy, June 2003. IEEE Computer Society Press.

[100] B. N. Shand. Trust for Resource Control: Self-Enforcing, Automatic, Rational Contracts Between Computers. Technical Report UCAM-CL-TR-600, University of Cambridge, Computer Laboratory, Aug. 2004. PhD Thesis.

[101] SPASS. An Automated Theorem Prover for First-Order Logic with Equality. http://spass.mpi-sb.mpg.de/index.html.

[102] M. Spiteri. *An Architecture for the Notification, Storage and Retrieval of Events*. PhD thesis, University of Cambridge, 2000.

[103] P. Steggles, P. Webster, and A. Harter. The Implementation of a Distributed Framework to Support Distributed Applications. In *PDPTA '98: Proceedings of the 1998 International Conference on Parallel and Distributed Processing Technique and Applications*, pages 381–388, Las Vegas, NV, July 1998. CSREA Press.

[104] M. Stringer, M. Eldridge, and M. Lamming. Towards a Deeper Understanding of Task Interruption. In *Proceedings of Workshop on Situated Interaction in Ubiquitous Computing at CHI '00*, pages 26–27, The Hague, The Netherlands, Apr. 2000.

[105] SAP: Systeme Andwendungen Produkte in der Datenverarbeitung. http://www.sap.com:80/company/index.aspx.

[106] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The Active Badge Location System. *ACM Transaction on Information Systems*, 10(1):91–102, 1992.

[107] C. Weidenbach. *The Theory of SPASS version 2.0*. Max-Planck-Institut fur Informatik.

[108] M. Weiser. Some Computer Science Problems in Ubiquitous Computing. *Communications of the ACM*, 36(7):75–84, 1993.

[109] L. A. Zadeh. Outline of a New Approach to the Analysis of Complex Systems and Decision Processes. In *Man and Computer*, Bordeaux, 1972. IEEE Computer Society Press.

[110] F. R. H. Zijlstra, R. A. Roe, A. B. Leonora, and I. Krediet. Temporal Factors in Mental Work: Effects of Interrupted Activities. *Journal of Occupational and Organizational Design.*, 72(2):163–185, 1999.