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

Eleftheria Katsiri[1] and Alan Mycroft[2]

[1] Laboratory for Communication Engineering, University of Cambridge,
[2] Computer Laboratory, University of Cambridge
William Gates Building, 15 JJ Thompson Avenue, Cambridge CB3 0FD, UK
`ek236@eng.cam.ac.uk` `am@cl.cam.ac.uk`

Abstract. We present a dynamic knowledge-base maintenance system for representing and reasoning with knowledge about the Sentient Computing environment. Sentient Computing has the property that it constantly monitors a rapidly-changing environment, thus introducing the need for abstract modelling of the physical world which is at the same time computationally efficient. Our approach 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 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. This is demonstrated through a prototype implementation.

1 Introduction

Our research is focused on *Sentient Computing* [11] applications. Sentient computing is the proposition that *applications can be made more responsive and useful by observing and reacting to the physical world*. Awareness comes through *sensing* and a *sensor infrastructure* (Fig. 1(a)), distributed in the environment, 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 perform easily complex computations involving spatial and temporal notions of a dynamic changing environment. E.g. 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. e.g. "*You asked me to remind you, to give Tom his book back, when you meet him.*" Sentient Computing applications can be viewed as a logical layer, namely, the *Application Layer* in the *Sentient Applications layered architecture* (Fig. 1(a)).

However, 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 get produced by the sensors and which constitute a *low-level, precise, knowledge layer.* To illustrate this using the previous example, the sensor that dispatches information about a user's position, only knows who the user is and his coordinates in space. 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.

We propose 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, which can in turn be used easily by the application layer. Furthermore, we believe that the proposed *deductive reasoning* does not compromise computational efficiency and performance even for very large distributed environments.

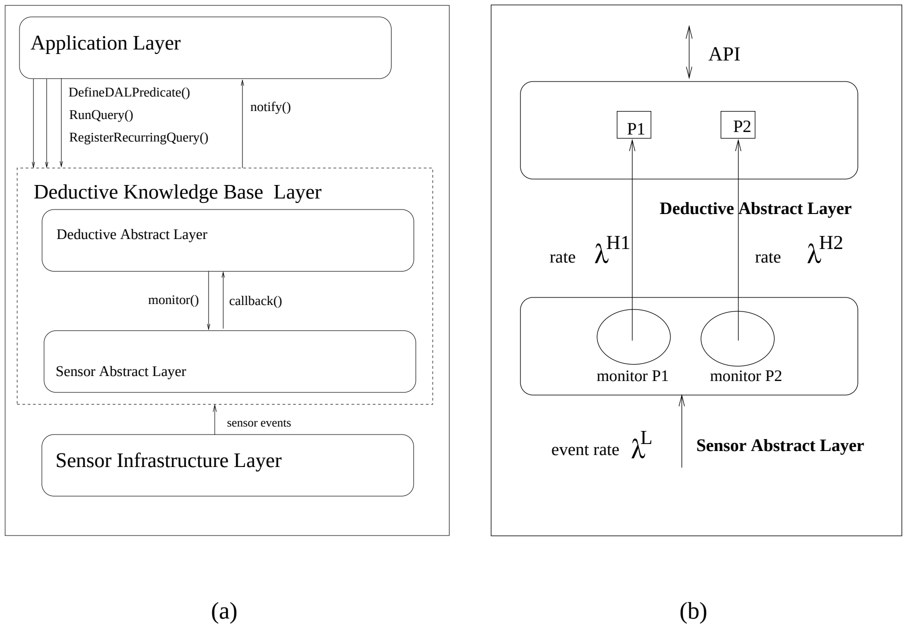

Fig. 1. The Sentient Applications layered architecture and its API (a) andthe dual-level architecture (b) .

This work 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*.

1.1 Layered Interfaces

For the abstract model of the Sentient world we chose to design a dual-layer knowledge representation architecture. Our design approach is inspired by the *OSI paradigm* [14] for layered network architecture where each layer incorporates a set of similar functions and hides lower-level problems from the layer above it thus achieving simplicity, abstraction and ease of implementation.

1.2 API

The application Layer communicates with the dual Deductive Knowledge base layer via an API consisting of a ***DefineDALPredicate()***, ***RunQuery()*** and ***RegisterRecurringQuery()*** interfaces (see Fig. 1(a)). 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.

The *RunQuery()* command is similar to an SQL SELECT statement in that it returns the current stored state of the Sentient world by first trying to obtain the information of interest from the Deductive Abstract Layer. If that piece of information is not contained there, a communication process between the two knowledge layers is triggered so that this piece of knowledge is generated from the low-level Sensor Abstract Layer and maintained dynamically in DAL before it is returned to the application layer. In this way, it will be available for future queries.

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 the *publish-subscribe* protocol.

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 [2]. 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 an update in the Deductive Abstract Layer accordingly, creating instances of the predicates that hold and destroying these that are no-longer true, as the case may be.

In this way, the two knowledge layers behave as a *two-level cache* for the Application Layer, enabling scalability.

1.3 Paper Outline

The rest of the paper is organised as follows: First, significant work in related areas is discussed and then, the architecture of the proposed system is described. Section 4 presents a formal definition of the knowledge representation architecture using first-order logic. Sentient Applications reason about the available knowledge through an interface which is described in Section 5. Next, a prototype implementation, discussed in Section 6, confirms the achieved scalability. Last, some conclusions and future work are discussed.

1.4 What is Not a Goal of this Paper.

We have chosen to describe the *deductive component* of the proposed architecture by using *first-order logic*. We believe first-order logic to be a very powerful and general tool for representing and reasoning with knowledge and we have found it to be appropriate and sufficient for the needs of Sentient Computing as is demonstrated in later sections. It is not our goal to provide a comparative study between first-order logic and other *reasoning schemes* such as *planning* and *description logics*, however a concise survey of literature related to planning and description logics can be found in section 2.

2 Related work

Having an efficient and scalable data model is of great importance for Sentient Computing, not only because of the *dynamic* and *rapidly changing* nature of the processed data in this paradigm, but also because of the *heterogeneity* of it. In this section we chose to survey literature in systems that are faced with similar concerns.

SPIRIT [10] is the most influential system in this area and is also the only system which tries to address scalability as an issue which is inherently linked with processing sensor-derived data. SPIRIT is a sophisticated system that 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. However, its approach towards both data-modelling and scalability is quite different from ours. 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. The indexing algorithm uses a quadtree for indexing spaces of arbitrary sizes and geometries and provides fast and predictable insertion, updating and query of spaces.

Since we believe *scalability* to be an *enabler for distribution*, it is useful to discuss the potential of efficiently distributing an architecture in order to cater for larger sizes of the physical world. In this aspect, although SPIRIT supports parallelism in the level of the storage of the world model objects [25], the distribution capabilities of the above mentioned *spatial indexing algorithm* are unclear. In fact, calculating relationships between spaces which are not stored on the same computer suggests a high communication overhead between the distributed elements which may affect significantly the response time of the algorithm.

Hence, SPIRIT uses a bottom-up, *engineering* approach in order to address scalability in calculating spatial updates in the world model.

FLAME [13, 3] is a development platform [13, 3] providing middleware support for applications by modelling location information. FLAME uses a Spatial Relation Model to assign regions to location events and generate *events* that denote region containment and overlapping.

Apart from the above mentioned development platforms, context-aware [24] applications include Teleporting [23], ComMotion [19], CyberMinder [1] and Proem [17]. The Teleporting System developed at Olivetti Research Laboratory is a tool for experiencing mobile X sessions. It provides a familiar, personalised way of making temporary use of X displays as the user moves from place to place. ComMotion is a GPS-based [4] system [19] which 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. CyberMinder is a system that delivers context-aware reminders, based on a specific set of contextual information such as location, time and agenda items. Proem is a peer-to-peer system, which matches pairs of mobile users according to their given profile of preferences. Last, Narayanan [20] presents the notion of grouping together locations that share spatial or logical features.

Our approach differs in that it tackles scalability in a top-down manner. It looks at the *stored data* through a *knowledge perspective*, and uses *logical deduction* in order to produce a mapping from the *heterogenous* data to different levels of abstract knowledge and exploiting the latter property in order to ensure efficient allocation of resources. By separating concerns between three key areas, namely, the sensor infrastructure that instruments the physical world, precise and abstract knowledge about the physical world and the application layer, we achieve a clearer view of abstract knowledge and provide good grounds for scalability in terms of knowledge queries as well as the creation of efficient interfaces. Furthermore, although our work is focused on the Sentient Computing environment, it is modelled in a formal way, which can be reapplied to all the above-described development platforms, independent of the underlying sensor technology, accommodating at the same time application systems such as [19], [1], [17] as users of the proposed architecture. For this reason, we adopted the formalism of *first-order logic*, as we believe this to be a powerful mechanism for reasoning. First-order logic, provides the desired expressiveness that applications need in order to define their requirements in knowledge. Several other alternatives, that may be applicable to *knowledge-representation and reasoning* in this area, are discussed briefly below:

Planning Systems [22] 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* 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.

Description Logics [21] 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 with 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*. Similarly to planning, any potential contribution of Description Logics to Sentient Computing is yet to be determined.

3 Knowledge Representation

This section describes the architecture (Fig. 1(b)) of the two logically distinct layers of knowledge representation and discusses their functionality as a two-layered cache for the Sentient application layer.

We use a definition introduced by Samani and Sloman in [18], according to which an *event* is a happening of interest that occurs instantaneously at a specific time. Furthermore, we define the *Sentient environment* to be the *physical* environment and the *current logical state* of the Sentient environment 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.

A particularly interesting source of events are the ones that characterise the *location* of an object. These are generated by a *location system* such as the Active BAT [10] where the position of users in 3-D space is 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 co-ordinates.

A more abstract view about the state of the Sentient environment can easily be *inferred* from the knowledge stored in the Sensor Abstract Layer. E.g. 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 that the user is present in, can also be inferred. [1]

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, the duration of each state, as well as when they stopped holding. 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 [16].

The two layers interact through a monitor-callback communication scheme. 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 alleviating 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 in a significantly lower rate $(\lambda_1^H {}_2\lambda^H)$ than it does in SAL (λ^L) ensuring in this way that any large amount of physical data can be processed by replicated SALs, maintaining at the same time the overall system scalability.

Example. In order to illustrate the functionality of the dual-layer architecture in more detail, let us consider the case where the Application layer is interested in receiving notification whenever two or more users are co-located. Through a *RunQuery()* 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 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 user's 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 are still co-located. If any of the co-located users exits 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, not the whole state of the Sentient Environment is changed, as most of abstract knowledge remain unaltered. Our approach in generating a new current state is similar to updating a table in a traditional database.

[1] 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.'

3.1 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 which can be made available to the application layer at a significantly low cost than if it would be generated directly from the Sensor Abstract Layer. There are three key reasons that enable DAL to act similarly to a fast cache for the application layer: The availability of the abstract knowledge in DAL, 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 1(b) depicts the different rates with which knowledge is updated in each layer. Section 5.2 discusses in more detail, computational concerns associated with the functionality of the two layers.

4 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.

4.1 First order logic

First order logic [6] 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 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 M 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 call a *knowledge base* K 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 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 has as a result 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. Nomenclature is taken from logic programming.

4.2 Naming convention for predicates

For reasons of clarity and simplicity, we adopt the following naming convention 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 which represent the beginning and wherever appropriate, the end of the situation they refer to. For the description of the predicates we have used a *named parameter notation* based on the CLIPS [26, 9] syntax. Table 1 portrays some significant predicates. Each *predicate argument* has an associated *value* which is denoted with *?argument-value*. The predicates and their arguments are discussed in detail in sections 4.3 and 4.4.

	Current Predicates	Historical Predicates
DAL	**(H_UserAtLocation**(uid $?uid$)(rid $?rid$) (start-time ?*start-time*))	**(H_ UserAtLocationHistoric**(uid $?uid$)(x $?x$)(y $?y$) (z $?z$)(start-time ?*start-time*)(end-time ?*end-time*))
	(H_UserColocation(uid-list $?uid_1 \cdots ?uid_n$) (rid ?*region-id*)(start-time ?*time-value*))	**(H_UserColocationHistoric**(uid-list $?uid_1 \cdots ?uid_n$) (rid ?*region-id*)(start-time ?*time-value*) (end-time ?*time-value*))
	(H_UserIsPresent(uid $?uid$)(start-time ?*time-value*))	**(H_UserIsPresentHistoric**(uid $?uid$) (start-time ?*time-value*)(end-time ?*end-time*))
SAL	**(L_UserAtLocation**(uid $?uid$)(x $?x$)(y $?y$)(z $?z$))	-

Table 1. Naming convention for logical predicates

4.3 Sensor Abstract Layer (SAL)

The knowledge base of this layer contains upto N facts of type *L_UserAtLocation(uid,x,y,z)* [2] that represent an object's last known position in 3-D space in terms of its Cartesian coordinates x, y, z. $L\_UserAtLocation$ is the most precise location known to the system for each user. The variable *?uid* represents the unique user identification for that particular user. In examples we use users' full names as identifiers. The variables *x,y,z* represent the user's last known co-ordinates. 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* each corresponding to the M_1 known atomic regions of physical space (e.g. rooms and polyhedral areas of space) and M_2 nested regions (floors, larger areas, buildings, neighbourhoods). There are therefore four distinct type of predicates represented in this layer.

(*L_UserAtLocation* (uid *?uid*)(x *?x*)(y *?y*) (z ?z))
(*L_AtomicLocation* (rid *?region-id*) (polyhedra $?n_1?n_2 \cdots ?n_j$))
(*L_NestedLocation* (rid *?region-id)* (site-list $?site_1 \cdots ?site_k$))
(*L_InRegion* (x *?x*)(y *?y*)(z ?z)(rid *?region-id*))

As the people move in space, a location system generates in average λ^L *L_UserAtLocation* facts/sec per mobile user and asserts them in the knowledge base. For each new fact of type *L_UserAtLocation* 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.

The predicate *L_AtomicLocation* associates a named location such as "Room 5" characterised by a unique identifier, the *region-id*, with a set of j points, $n_1 \cdots n_j$, which form the nodes of a polyhedral region that defines that area.[3] 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, which determines the *smallest* region that contains the given co-ordinates, as expressed in the *L_UserAtLocation* predicate.

4.4 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 the knowledge of the accuracy of the exact user position (as this is only known

[2] The positional parameters notation *L_UserAtLocation(uid,x,y,z)* is used interchangeably throughout this paper with the named parameter notation *L_UserAtLocation*(uid *?uid*)(x *?x*)(y *?y*) (z *?z*)) for simplicity reasons.
[3] We assume a co-ordinate system that assigns a set of co-ordinate values *x,y,z* to each position in space.

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 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 accumulates gradually information about what has happened in the Sentient world. Now, the format of the DAL predicates is discussed in detail.

The DAL Current predicates. *DAL current* predicates describe a situation which occurred at an instant t_0 and which still holds at the current instant which is represented with the value *now*. Such predicates have the 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_UserAtLocation*(uid "Mary") (rid "coffee-machine-area") (start-time 11:02))

(*H_UserAtLocation*(uid "James") (rid "floor-4") (start-time 13:05))

where *?uid* represents the user's unique identification and *?region-id* 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 the 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 $?uid_1\cdots?uid_n$)(rid *?region-id*)(start-time *?time-value*))

This process is explained in more detail in section 5.2.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 which occurred at a time instance t_0, remained holding for a duration d and ceased holding at a time instance 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 he was in. E.g. 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 pm, is expressed below: It is worth noting that there can be multiple instances of historic predicates for the same user.

(*H_UserLocationHistoric* (uid-list "Jane Hunter")(rid "Meeting Room")(start-time 12.46) (end-time 12.55))

(*H_UserCoLocationHistoric* (uid-list "Mike Smith" "Jane Hunter")(rid "Meeting Room")
(start-time 12.46) (end-time 12.55))

4.5 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. DAL predicates get created through the *DefineDALPredicate()* API call (see section 1.2) and *instances* of these predicates (facts) get generated from the Sensor Abstract Layer by the *monitor()* and *callback()* calls (see Section 1.2).

5 Queries

Queries are used by the application layer in order to capture and return the current instance of the stored knowledge about the Sentient World. *Queries* are similar to SQL [5] SELECT statements in the theory of relational databases. We can view a ***query*** as a first-order logical expression $f(c_1, c_2 \cdots c_n)$ which 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 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. We have chosen to adopt the *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 1.2).

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 [1]. Equally interesting as an example is the case where a user enters a conference site and is interested to know if there is somebody present from the University of Cambridge.

The number of conditions in the queries depends on the underlying knowledge base model. E.g. 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_UserAtLocation*(uid *?uid*) (x *?x*) (y *?y*) (z ?z))$\wedge$
(*L_AtomicLocation*(rid ?region-id) (polyhedron $?n_1 ?n_2 \cdots ?n_j$))$\wedge$
(*L_InRegion*(x ?x) (y ?y)(z ?z) (rid *?region-id*))

The “|” operator is similar to an SQL *SELECT* operator in that it returns the values for the associated variables. In this case, *uid* represent 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 which can be characterised by the co-ordinates x,y,z.
– The system must know of at least one region with id rid which can be characterised by a known polyhedral shape, and
– 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 it would assume a simpler form:

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

$uid|$(*H_UserIsPresent*(uid *?user-id*) (start-time *?t*))

Query 1 and Query2 are defined to be equivalent directly by the application layer through the RegisterRecurringQuery interface and its arguments. E.g. for the case of the (*H_UserIsPresent*(uid *?uid*) (start-time *?t*)) predicate, an application would have to issue the following statement:

RegisterRecurringQuery(*application identity*, (*H_UserIsPresent*(uid *?uid*) (start-time *?start-time*)), Query 1, Query 2)

In the above statement, the argument *application identity* describes an identification for the application that has issued the statement and to which the *answer* will be returned to. The *H_UserIsPresent* predicate is similar in content to the *H_UserAtLocation* one and it is useful as an abstraction of the user's location, where the actual location is of no interest to the application.

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_UserAtLocation*(uid $?uid_1$)(x $?x_1$)(y $?y_1$) (z $?z_1$)$\wedge$
(*L_UserAtLocation*(uid $?uid_2$)(x $?x_2$)(y $?y_2$) (z $?z_2$))$\wedge$
(*L_AtomicLocation*(rid ?region-id) (polyhedron $?n_1?n_2 \cdots ?n_j$))$\wedge$
(*L_InRegion*(x $?x_1$)(y $?y_1$)(z $?z_1$) (rid ?region-id))$\wedge$
(*L_InRegion*(x $?x_2$)(y $?y_2$)(z $?z_2$)(rid ?region-id))$\wedge$
(*L_InRegion*(x_2, y_2, z_2, rid))$\wedge$
($uid_1 \neq uid_2$)

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 $?uid_1?uid_2$)(rid *?rid*)(start-time *?t*))

Similarly, Queries 3 and 4 are declared to be equivalent through the RegisterRecurringQuery interface:

RegisterRecurringQuery(*application identity,* (*H_UserCoLocation* (uid-list $?uid_1?uid_2$)(rid *?rid*) (start-time *?t*)), Query 3, Query 4)

Based on the RegisterRecurringQuery definitions, Queries 1 and 3 are equivalent to Queries 2 and 4 respectively. However, Queries 2 and 4 have in 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 discusses in detail the effect of the above observation to 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).

5.1 Recurring queries

A second approach for the application layer to derive information from the knowledge base is by *registering interest* to a recurring situation that gets 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 the *publish-notify* protocol for distributed systems, in registering interest in receiving information about the *future* occurrences of the situation in question.

E.g. 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.” We can view a ***recurring query*** as a first-order logical expression $f(c_1, c_2 \cdots c_n)$ which 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. Such an event can be a Supervisor Alert event which is defined elsewhere in the system. When it 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, where upon satisfaction of the query, a notification action is being performed. E.g. “Whenever my supervisor enters the lab, notify me.” Recurring queries can be expressed as *logical implications*, in which 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)*

$$uid|(\textit{H\_UserIsPresent}(\text{uid “Andy Hopper”}) \Rightarrow \text{notify(Supervisor Alert)})$$

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.

5.2 Analysis

Having discussed queries and recurring queries, we can now look into how the two-layer knowledge scheme ensures scalability .

In order to illustrate this, we consider a prototype implementation, where queries and recurring queries are implemented in each layer by means of a CLIPS [26] inference engine. Each query is mapped to a CLIPS rule. CLIPS implements a forward chaining rule interpreter that given a set of *rules* applied on 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* [8] is used for this purpose.

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

– First, as can be seen from Sections 5, 5.1, *queries* that are executed in the Deductive Abstract Layer such as Query 3, assume a *much simpler form* than those executed in the Sensor Abstract Layer (Query 1), as the latter have *more conditions in 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 1(b)). 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, the machine that hosts DAL has fewer real-time constraints, 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 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 5) 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 lower number of computational steps.

We have implemented the proposed architecture using the Jess[12] production system. Jess is a java-based implementation of CLIPS. For the acquisition of real-time location information, we have built a middleware component [15] that interfaces the Active BAT system using CORBA structured events, and translating them into Jess facts.

6.1 Sensor Abstract Layer.

A model was created in Jess for the LCE [7] 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. An instance of the lower-layer was captured and 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).*

$$\begin{aligned}&(uid_1, uid_2)|\ (\textit{L\_UserAtLocation}(\text{uid } ?uid_1)(\text{rid ?rid}))\wedge\\&(\textit{L\_UserAtLocation}(\text{uid } ?uid_2)(\text{rid ?rid}))\wedge\\&(\textit{L\_AtomicLocation}(\text{rid ?region-id})\ (\text{polyhedron } ?n_1 ?n_2 \cdots ?n_j))\wedge\\&(uid_1 \neq uid_2)\end{aligned}$$

The Rete Algorithm Our implementation uses the Rete Algorithm [8] 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 in the knowledge base, or whenever a new fact s asserted or retracted. More specifically, for each predicate included in the query, the network creates a one-input node, portrayed in *red* in Fig 2. Node n_1 corresponds to the predicate *L_UserAtLocation* and node n_2 to the predicate *L_AtomicLocation*. Node n_3 corresponds to the condition $(uid_1 \neq uid_2)$. Also portrayed in red is the root node of the network, n_0. A two-input *(green)* node is created for each conjunction of predicates. Node n_5 corresponds to the conjunction:

(*L_UserAtLocation*(uid $?uid_1$)(rid $?rid$))$\wedge$ (*L_UserAtLocation*(uid $?uid_2$)(rid $?rid$)) Node n_6 corresponds to the conjunction:

$$\begin{aligned}&(\textit{L\_UserAtLocation}(\text{uid } ?uid_1)(\text{rid } ?rid))\wedge\\&(\textit{L\_UserAtLocation}(\text{uid } ?uid_2)(\text{rid } ?rid))\wedge\\&(\textit{L\_AtomicLocation}(\text{rid } \textit{?region-id})(\text{polyhedron } ?n_1 \cdots ?n_j))\end{aligned}$$

Node n_7 is also a two-input node, that represents the conjunction of the above predicates with the condition $(uid_1 \neq uid_2)$. Finally, node n_8 is a terminal node that determines whether the query is satisfied or not.

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 which 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_1 checks whether any of the received tokens correspond to facts of type *L_UserAtLocation*[4] and passes all such tokens to node n_5. Node n_5 checks all UserAtPosition tokens against each other, in order to determine which pairs satisfy the conjunction:

(*L_UserAtLocation*(uid $?uid_1$)(rid $?rid$))$\wedge$ (*L_UserAtLocation*(uid $?uid_2$)(rid $?rid$))

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

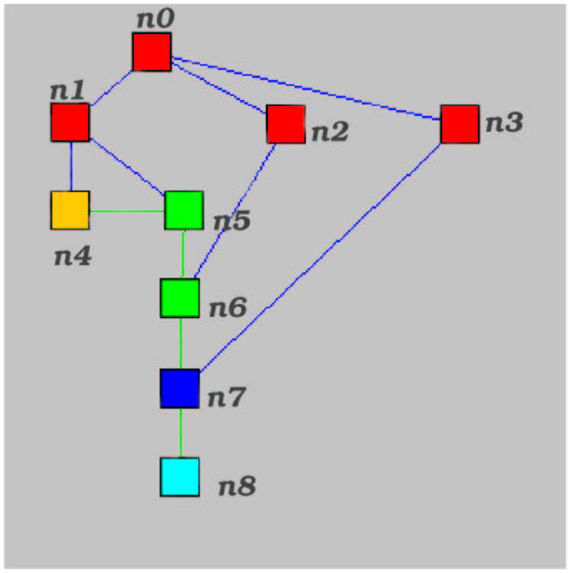

Fig. 2. The Rete Network for the Return All Co-Located Users query (Sensor Abstract Layer)

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 :

$$\begin{array}{l}(\textit{L\_UserAtLocation}(\text{uid } ?uid_1)(\text{rid } ?rid))\wedge \\ (\textit{L\_UserAtLocation}(\text{uid } ?uid_2)(\text{rid } ?rid))\wedge \\ (\textit{L\_AtomicLocation}(\text{rid } ?\textit{region-id})(\text{polyhedron } ?n_1 \cdots ?n_j))\end{array}$$

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 get an measure of the computational complexity that the implemented scheme entails, we chose to look at the number of node activations and the number of tests performed in total by the nodes on the network. The results are shown in Table 2 (SAL).

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

Table 2. Pattern Matching Costs.

6.2 Deductive Abstract Layer

Repeating the previous experiment, with the same initial state, we now consider Query 4 (see Section 5) which is executed in the Deductive Abstract Layer. The network for this query is portrayed in Fig. 3.

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 2. It is worth noting that the number of computational steps executed by the Rete algorithm for pattern matching each query are

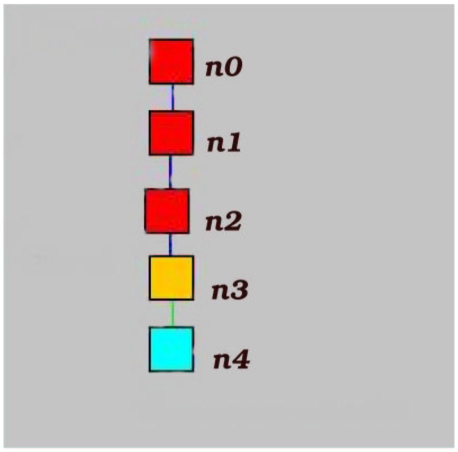

Fig. 3. The Rete Network for the Return All Co-Located Users query (Deductive Abstract Layer)

significantly lower for the DAL query. Taking into consideration that both networks (see Fig. 2, Fig.3) behave similarly to acyclic finite automata, which are triggered repeatedly each time a fact is asserted or retracted in each knowledge layer respectively, we can easily infer 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. Last, SAL is continually *interrupted* by a very high event rate which has an immediate effect on the machine that hosts that layer.

7 Conclusions and future work

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 proved suitable for Sentient Computing, especially in the context of the proposed architecture which is based on a cache-like, dual-layer scheme which maintains abstract knowledge in the higher 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 interface. Such predicates are contained in the Sensor Abstract Layer and by having only threshold changes reflected at the Deductive Abstract Layer. Maintaining abstract knowledge is a requirement of the Sentient Application layer and it is made available to Sentient Applications through a mechanism of queries which are mainly executed at the Deductive Abstract layer. Experiments with a prototype implementation confirm that the two-layered architecture is more efficient than a single-layered one. Future work will involve designing and implementing a large-scale, fully distributed architecture based on the proposed system.

8 Acknowledgements

We are grateful to Andy Hopper and Mike Gordon for discussion and encouragement.

References

1. Dey K Anind and Gregory D.Abowd. CyberMinder: A Context-aware System for Supporting Reminders. In *Proceeding of CHI 2000*. CHI2000, 2000.
2. George Coulouris, Tim Kindberg, and Jean Dollimore. *Distributed Systems: Concepts and Design*. Addison-Wesley, 2001.

3. George Coulouris, Hani Naguib, and Scott Mitchell. Middleware Support for Context Aware Multimedia Applications.
4. P. Dana. Global Posioning System Overview", Department of Geography University of Texas at Austin. http://www.colorado.Edu/geography/gcraft/notes/gps/gps.html, 1998.
5. C. J. Date and Hugh Darwen. *A Guide to the SQL Standard, Third Edition*. Addison-Wesley Publishing Company, Inc., 1993.
6. Elliott Mendelson. *Introduction to Mathematical Logic*. Wadsworth and Brooks Cole Advanced Books Software, 1990.
7. Laboratory for Communications Engineering (LCE). http:://www-lce.eng.cam.ac.uk.
8. Charles L. Forgy. Rete: A fast Algorithm for the Many Pattern/many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.
9. Joseph C. Giarratano. *CLIPS User's Guide, Version 6.10*, chapter ch. 8. unknown, 1998.
10. Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster. The Anatomy of a Context-Aware Application. *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking August 15-19,1999, Seattle*, 1999.
11. Andy Hopper. The Royal Society Clifford Paterson Lecture: Sentient Computing, 1999.
12. http://herzberg.ca.sandia.gov/jess. Jess, the Java Expert System Shell.
13. http://www lce.eng.cam.ac.uk/qosdream/. QoSDREAM: Quality of Service for Distributed REconfigurable Adaptive Multimedia.
14. http://www.acm.org/sigcomm/standards/iso_stds/OSI_MODEL. OSI 7498, open System Interconnection Model.
15. D. Ipina and E. Katsiri. A Rule-Matching Service for Simpler Develpment of Reactive Applications. In *Middleware 2001*. http://computer.org/dsonline/0107/features/lop0107.htm, November 2001.
16. E Katsiri. Principles of Context Inferences. In *Ubicomp 2002 Additional Proceedings*, 2002.
17. G. Korteum, Z. Segall, and T.G. Thomson. Close Encounters: Supporting Mobile Collaboration through Interchange of User Profiles. In *HUC'99*, pages 171–185, 1999.
18. Masoud Mansouri-Samani and Morris Sloman. Gem: A Generalised Event Monitoring Language for Distributed Systems. *Distributed Systems Engineering Journal*, Vol. 4(No. 2), June 1997.
19. N. Marmasse. comMotion. In *CHI'99*, pages 320,321, 1999.
20. Ajith K. Narayanan. Realms and States: A Framework for Location Aware Mobile Computing. In *Workshop on Mobile Commerce (MOBICOM)*, 2001.
21. Danielle Nardi and Ronald J. Brachman. An Introduction to Description Logics. Published on the Internet.
22. Nils J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, 1998.
23. Tristan Richardson. Teleporting - Mobile X Sessions. In *Proceedings Ninth Annual X Technical Conference, Boston MA, Technical Report 95.7*, 1995.
24. Bill Schilt, Norman Adams, and Roy Want. Context-Aware Computing Applications. *tbd*, 1994.
25. Pete Steggles, Paul Webster, and Andy Harter. The Implementation of a Distributed Framework to Support Distributed Applications. Technical report, The Olivetti and Oracle Research Laboratory, 1998.
26. www.ghg.net/clips/CLIPS.html. CLIPS: A Tool for Designing Expert Systems.