Representing and Reasoning with Situations for Context-Aware Pervasive Computing: a Logic Programming Perspective

Seng W. Loke
School of Computer Science and Software Engineering,
Monash University, Australia
swloke@csse.monash.edu.au

August 18, 2004

Abstract

Context-aware pervasive systems are emerging as an important class of applications. We present a declarative approach to building context-aware pervasive systems, and introduce the notion of the situation program which highlights the primacy of the situation abstraction for building context-aware pervasive systems. We show how to manipulate situation programs using meta-programming within an extension of the Prolog logic programming language which we call LogicCAP. Such meta-reasoning enables complex situations to be described in terms of other situations. We also discuss how the design of situation programs can affect the properties of a context-aware system. The approach encourages a high-level of abstraction for representing and reasoning with situations, and supports building context-aware systems incrementally by providing modularity and separation of concerns.

1 From Context to Situations

Context-aware computing has enjoyed remarkable attention from researchers in diverse areas such as mobile computing (Schilit *et al.*, 1994) and human

computer interaction (Moran and Dourish, 2002). It is also an important idea explored in connection with pervasive computing and ambient intelligence.[1]

The notion of context itself is not new and has been explored in areas such as linguistics, natural language processing, philosophy, AI knowledge representation and problem-solving, and theory of communication (Akman, 2002; Bouquet *et al.*, 2003; McCarthy, 1993; Brezillon, 2003). In such work, context is given focus and primacy (e.g., treated as first class objects in a logic) enabling assertions to be made about contexts and context to be explicitly reasoned about in applications.

The Free On-line Dictionary of Computing[2] defines context as

> *"that which surrounds, and gives meaning to, something else."*

Such a definition might be instantiated according to the need. Whether that 'something' is an assertion in a logic, an utterance, or a computer system, with an appropriate definition for 'meaning', the intuition captured by the word 'context' serves its purpose. The work by (Schilit *et al.*, 1994) provides an instantiation of that definition, from the perspective of distributed, mobile, and ubiquitous computing (or pervasive computing[3]): a person is that something and context refers to information about a person's proximate environment such as location and identities of nearby people and objects. In (Dey, 2001) is an operational (and arguably broader) definition of context:

> *"Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."*

Context-aware applications aim to use such contextual information to do the right thing at the right time automatically for the user. There has been much work in identifying what such information can be, the structure of the information, how to represent such information, and how to exploit it for a specific application. Such work might focus on a specific kind of contextual information such as location models,[4] world models (e.g., Lehmann *et al.*, 2004) and

[1] See the symposium at http://www.eusai.net

[2] Access via http://www.dictionary.com

[3] One view of pervasive computing is as a combination of mobile computing and ubiquitous computing.

[4] For example, see http://research.microsoft.com/workshops/UbiLoc03/

activity models (e.g., Muhlenbrock *et al.*, 2004; Koile *et al.*, 2003; Tapia *et al.*, 2004),[5] or identify characteristics of contextual information (Henricksen *et al.*, 2004).

Pervasive computing utilizes contextual information about the physical world. Hence, the connection of sensor information to context-aware pervasive computing is clearly important (Yoshimi, 2000; Barkhuus, 2003; Hopper, 1999; Patterson *et al.*, 2003), and relates to what can be sensed, the best way to acquire sensor information, and how to reason with sensor information to infer context. In fact, any information which can be practically obtained via sensors can be used as context including the emotional states of users (Picard, 1997) and movements (Headon, 2003). Where the entity is an artifact instead of a person, we have context-aware artifacts.

There is tremendous variety and diversity in what can be context, and the way context can be acquired and modelled, and this is an avenue of much interest and research. Recent workshops[6] have focused on just this topic. Given the challenges in representing, structuring, managing and using context, it is not surprising that various knowledge representation formalisms and techniques have been applied, ranging from ontologies (Chen *et al.*, 2004b; McGrath *et al.* 2003; Wang *et al.*, 2004; Matheus *et al.*, 2003) (that can provide concepts for describing context and enable reasoning with and reuse of contextual information), first-order logic theories (Katsiri and Mycroft, 2003; Ranganathan and Campbell, 2003), to conceptual graphs (Peters and Shrobe, 2003). Such work, however, is not simply a return to previous AI knowledge representation about context, but consider what aspects to sense of the physical world for a given application and how best to represent such aspects, how to reason with sensed information, and the software engineering of context-aware pervasive systems.

Related to the notion of context is the notion of situation. The relation-

[5]Activity typically refers to some action or operation, undertaken by a human being, such as 'bathing', 'doing laundry', 'toileting', 'preparing breakfast', and 'listening to music', and so differs from situation. Perhaps one could conceive of a person in the state of preparing breakfast as a situation. However, in general, 'activity' and 'situation' are clearly not interchangeable, and we consider activity as a type of contextual information which can be used to characterize the situation of a person (e.g., preparing breakfast means the person is busy or has just woken up).

[6]Workshop on Context Modeling and Reasoning (CoMoRea 2004), Workshop on Modelling and Retrieval of Context (MRC 2004) (http://mrc2004.wysart.de/), and Workshop on Advanced Context Modelling, Reasoning and Management 2004 (http://pace.dstc.edu.au/UbiComp2004_ContextWorkshop.html)

ship between context and situation is illustrated in the above operational definition. A definition of situation from the American Heritage Dictionary[7] is as follows:

> *"The combination of circumstances at a given moment; a state of affairs."*

Besides context, Dey (2001) also defines situation, as follows:

> *"a description of the states of relevant entities."*

Hence, the idea is of aggregating (perhaps varieties of) context information in order to determine the situation of the entities (relevant to an application). In this sense, the situation might be thought of as being at a higher level of abstraction than context.

In philosophy and AI, much thought has gone into the idea of the situation, such as in situation theory (Barwise, 1989; Barwise *et al.* 1991) and the situation calculus. This perspective considers the primacy of the situation abstraction and noted that an agent (e.g., human) is able to individuate a situation. According to Devlin (1991), a situation is a

> *"structured part of reality that it (the agent) somehow manages to pick out",*

by "direct perception of a situation, perhaps the immediate environment, or thinking about a particular situation," and "individuation of a situation by an agent does not (necessarily) entail the agent being able to provide an exact description of everything that is and is not going on in that situation." In the case where the agent is a context-aware computer system, one can utilize the situation as a programming abstraction represented in some formalism to refer to something humans (e.g., the programmer) might naturally individuate. The idea is that the situation abstraction allows one to effectively "carve the world up" into manageable pieces which a collection of sensors of a system might recognize and respond to. It might also be possible to compose such pieces to construct more complex models of situations.

Our working hypothesis is that, in the software engineering of context-aware pervasive systems, the situation abstraction is useful. Not only can the system developer naturally individuate and identify situations for an

[7] Accessed from http://www.dictionary.com

application, but thinking at the level of situations provides a high level of abstraction for the developer.

In this paper, we present a novel way of representing situations taking into account the structure of a context-aware system as comprising sensors at one level and inference procedures to reason with context and situations at another level. We also consider how to manipulate situations as first-class entities and how to reason with our representation of situations within a programming language. We clearly differentiate between sensor readings, context, and situation in our model. Our model is also declarative and based on logic programming ideas. The model provides a basis for programming context-aware pervasive systems that emphasizes the primacy of the situation abstraction. Our approach encourages the separation of the different concerns in building a context-aware pervasive system including representing situations, representing context, relating context to sensors, building sensors, and reasoning with situations.

Constructing context-aware pervasive systems is generally a complex task and involves knowledge engineering, sensor data analysis, inferencing, and application programming. A broader aim of this work is to provide insight into systematically designing and constructing such systems. We contend that a key abstraction is the situation.

The rest of this paper is organized as follows. Section 2 presents how we represent situations using logic programs which we call *situation programs*. We give a denotational semantics of situation programs, aiming to say more precisely what a situation program is and what we intend to represent using it. In Section 3, we proceed to embed situation programs within a logic programming language, and provide programming examples. Section 4 discusses the relationships between the situations a system designer would like to model and the situation programs of a system, formulating useful properties of a context-aware system. Section 5 discusses related work and we conclude in Section 6.

2 A Logic Programming Approach to Characterizing Situations

There are many ways to describe a situation depending on the application. Here, we involve *sensors* in our definition of situation. We use a broad defini-

tion of sensor, which is taken to mean not only temperature, heat or motion sensors but any device or mechanism that is used to provide contextual information. This means that a positioning engine (which provides location information about a device and user) is also called a sensor. The clock in the computer with its associated operating system call can be considered a sensor if it used to return time which is used as contextual information in an application.

We provide a characterization of a situation by observing that a situation (when it occurs) effectively *imposes constraints on the output or readings that can be returned by sensors*, i.e. if S is the current situation, we expect the sensors to return values satisfying some constraints associated with S. We also represent such constraints as a logic program.

Our rationale is that, given a set of sensors, and a situation, when the sensors are working within that situation, the readings will necessarily fall within certain values. For example, if the situation is that a person (called Kenny, say) is attending a seminar in the university grounds, it is necessary that his location be that of some seminar room in the university, and that there are other people in the seminar room (e.g., a weight sensor in the floor of the room would give a reading larger than that of Kenny's weight). Hence, the location sensor's reading is constrained and so is the room's weight sensor given the occurrence of the situation. The necessary conditions are knowledge engineered into the system so that the situation is represented in the system. Different situations can then be characterized by different constraints imposed on the sensors. Two situations differ if they impose different constraints on the sensors - which implies that two situations might be indiscernible to the system given that they result in the same sensor readings, and in our case if they impose the same constraints on sensor readings. This way of characterizing situations provides a means to determine at any given time, what situation is being sensed, namely, the current situation (that is being sensed) is any one situation whose constraints are satisfied by the sensor inputs at that time. It is either the case that several such situations are equally valid simultaneously or only one (or some) are - knowledge must be engineered into the system to determine which is the case.

2.1 The Situation Program

Let each sensor be represented by a *sensor predicate*[8] of the form:

$$\langle \texttt{sensor\_id} \rangle \texttt{*(} \langle \texttt{inputs} \rangle \texttt{,} \langle \texttt{output readings} \rangle \texttt{).}$$

The output from a sensor is represented by a variable, and inputs to sensors by parameters. Then, a situation program S is defined as a collection of rules (or a logic program), which we call a *situation program*, each rule of the form:

$$\texttt{if } \mathcal{A} \texttt{ then } \mathcal{G}$$

where $\mathcal{G}$ is given by

$$\mathcal{G} \ ::= \ \mathcal{A} \mid \mathcal{S} \mid (\mathcal{G}, \mathcal{G}) \mid S\texttt{*>}E$$

$\mathcal{A}$ is an atomic goal formula (an ordinary Prolog-style term), $\mathcal{S}$ is a sensor predicate, "," denotes conjunction, S is a situation identifier and E is an entity (e.g. user, device, or software agent) identifier. We call the operator "in-situation" denoted by "`*>`". A goal of the form S`*>`E, read as a query "E in situation S?", is a meta-level goal that succeeds if *the contextual information about E is provable from S* in the way we describe later. Because S represents clauses (facts and rules that would hold) about the situation, the intuition of this operator is that E is in the real world situation represented by S if the contextual information about E holds in S. This is analogous to the supports operator "$\models$" in situation theory (Tin and Akman 1994; Barwise, 1989), where a situation s supports an infon i (representing a piece of information) is denoted by $s \models i$, except here we have a computational interpretation, so that a situation supports an infon if the infon (represented by a goal) is validated by the situation (represented by a logic program). There must also be at least one distinguished rule (which we call the *situation rule*) whose premise is a predicate naming the situation and, optionally, have a parameter denoting the entity. Also, we assume that the premises of the rules within a situation program are all different - we do not deal with disjunction; we return to this restriction in Section 6. The idea is that the collection of rules specifies constraints on the sensors' readings, analogous to a logic program in constraint logic programming languages where constraints are essentially relations on the variables.

[8] "Predicate" as in first order logic: http://mathworld.wolfram.com/First-OrderLogic.html

The rules of a situation program permit natural expression of explanation capabilities of a situation, i.e. if a situation occurs, then certain conditions and constraints should hold. As an example, we can define a `in_meeting_now` situation as follows. The sensor predicates are `location*(E,L)` which returns the location of an entity `E` in variable `L`, `diary*(E, Event, entry(StartTime, Duration))` which returns diary entries for entity `E` for a matching `Event`, `people_in_room*(L,N)` which returns the number of people at a location, and `current_time*(T)` which takes no inputs and returns the current time in a variable. The constraints the situation imposes on such sensors' readings can then be modelled by the following logic program:

```
if in_meeting_now(E) then
 with_someone_now(E),
 has_entry_for_meeting_in_diary(E).
if with_someone_now(E) then
 location*(E,L),people_in_room*(L,N), N > 1.
if has_entry_for_meeting_in_diary(E) then
 current_time*(T1),
 diary*(E,'meeting',entry(StartTime,Duration)),
 within_interval(T1, StartTime, Duration).
```

The program is viewed as a constraint in the sense that if the entity is in that situation, various relationships as specified above should hold. We could have written the rules as follows but it is less readable:

```
if in_meeting_now(E) then
 location*(E,L),
 people_in\_room*(L,N), N > 1, current_time*(T1),
 diary*(E,'meeting',entry(StartTime,Duration)),
 within_interval(T1, StartTime, Duration).
```

By using several rules, we can define constraints in a more readable style.

Our syntax of rules allow situation programs that refer to other situation programs. The above program might be rewritten as follows.

```
if in_meeting_now(E) then
  with_someone_now*>E,
  has_entry_for_meeting_in_diary*>E.
```

where `with_someone_now` is a situation with its own situation program containing the rule `if with_someone_now(E) then location*(E,L),people_in_room*(L,N), N > 1`, and similarly, the situation `has_entry_for_meeting_in_diary`. The advantage of being able to split rules into separate situation programs is modularity which encourages reuse.

We consider this question: given current sensor readings, is guessing the current situation based on these readings the inference process of deduction or abduction?[9] A different way of writing the rules that relate sensor readings with situations is as below, where sensor readings are stated as sufficient conditions for situations. For example:

```
if location*(E,L) and people_in_room(L,N)
  and N > 1 then with_someone_now(E)
```

The above rule states that if the sensors return certain readings, then the situation occurs. But the sensors could not have caused nor explain the situation. It is more accurate to say that if the sensors observed certain values, we would like to find an explanation or cause for the observations, i.e., in our case, what situation led to such sensors' observations. There may be more than one situation that would lead to the same observations. Then, determining which situation is most likely is required. The analogy is with diagnostic systems, where we try to find or guess the cause of an observed fault - it is the cause that determines what is observed. It is the occurrence of a situation that causes the sensors to observe certain values (i.e. have certain readings), rather than conversely, or in other words, the occurrence of a situation is an explanation for the sensor readings that are being obtained. Hence, we write rules where the situation is the premise and the expected sensor readings are the conclusions, capturing the knowledge that if a situation occurs, the sensors should have certain values.

2.2 A Denotational Semantics for Situation Programs

A designer of a system creates a situation program as a representation of a situation the system should be made to recognize. This subsection says

[9] Abduction refers to the process of inferring case(s) from rules and an observed result, i.e. computing explanations for observations, whereas deduction is inferring a result from rules and cases.

more precisely in what way a situation program represents situation(s) by providing a denonational semantics for situation programs.

We use the notion of the Herbrand model (Lloyd, 1984). Let $\mathcal{L}$ be a first-order language with a set of constants. The *Herbrand base* of $\mathcal{L}$ is the set of all ground atomic formulae of $\mathcal{L}$. A *Herbrand interpretation* for $\mathcal{L}$ is a subset of the Herbrand base of $\mathcal{L}$.

We consider a situation to be a part of the way the world happens to be (Barwise, 1989). Since we are dealing with logic programs, the inputs as provided to the sensor predicates by sensors can be treated as ground atomic formulae. More precisely, one could think of a situation (occurring as sensed by the system) as assigning Herbrand interpretations to the situation programs (stored in the system), for the sensor predicates; we say that the situation *yields* the Herbrand interpretation (by the process of sensing). A *perceived situation*[10] s, can be thought of as the function $s : SituationPrograms \rightarrow \mathcal{P}(HB)$, where HB is a Herbrand Base (for some suitably encompassing language $\mathcal{L}$).

A perceived situation is said to *satisfy* a situation program if it yield inputs to the sensors referred to in the situation program in such a way that the constraints (as represented by the situation program) are satisfied. We make this more precise. Let sit_p be the (ground) predicate in the premise of the situation rule, then a perceived situation is said to *satisfy* a situation program P if and only if the situation yields a Herbrand interpretation H, for P, which grounds P and is a model for P. Stated differently, the situation provides the sensor readings in such a way that sit_p can be inferred by abduction.

Given a context-aware system Γ with certain sensors and situation programs, we define the meaning of a situation program P (with respect to Γ), denoted by $[\![P]\!]_\Gamma$, as the set of all perceived situations that satisfy P.

One could think of a situation program as a *situation type* (in the situation-theoretic sense (Barwise and Perry, 1983)), and the process of a context-aware system attempting to understand an actual situation (occurring in a given time and sensed) is that of finding matches between the sensed situation and situation programs (stored in the system). Each situation program divides up the situations that the system can sense, according to whether the situation satisfies the constraints on sensor readings encoded

[10] A perceived situation is analogous to the idea of *scenes* in (Barwise, 1989), which are visually perceived situations. In our case, it is the system and not a human which is perceiving the situation.

in the situation program.

2.3 Executing Situation Programs: Evaluating S*>E

Because we represent situations as explanations for observations, we describe the procedure for evaluating the in-situation goal by forward-chaining over rules in situation programs. It is similar to the standard forward-chaining reasoning algorithm and follows the definition of the *satisfy* relation between perceived situations and situation programs described in the previous subsection. This algorithm effectively infers the constraints, and at the same time, checks if the sensor readings satisfy the constraints: (1) Initialise the set of inference rules and goals (i.e. assumptions and constraints in this case). (2) Determine which inference rules are applicable given the current set of goals. If no rules are applicable, evaluate the goals. If several rules are applicable, we use all the rules - the goals inferred from the conclusion of each rule is added. (3) Apply the rule to infer one or more new goals. (4) Repeat from step 2. For the example in Section 2.1, the algorithm will execute in this way for the goal `in_meeting_now*>john`:

1. Initialise the set of inference rules and goals. We start with the predicate describing the situation (i.e., `in_meeting_now(`E`)`, with E instantiated to `john`) as the initial goal, i.e. we basically assume the situation and then explore its implications by forward chaining on the rules.

2. Determine which inference rules are applicable given the current set of goals. Starting from the initial assumption, the only rule applicable is the first one.

   ```
   if in_meeting_now(E) then
     with_someone_now(E),
     has_entry_for_meeting_in_diary(E).
   ```

3. Apply the rule to infer one or more new goals. In this case, we infer `with_someone_now(john)` and `has_entry_for_meeting_in_diary(john)`.

4. (Repeat from step 2 of the algorithm.) Starting from `with_someone_now(john)` and `has_entry_for_meeting_in_diary(john)`, the applicable rules are the next two as shown earlier.

5. Now, we infer the constraints on the sensor readings from each rule: (a) `location*(john,L)`, and `people_in_room*(L,N), N > 1`, and (b) `current_time*(T1)`, `diary*(john, 'meeting', entry(StartTime, Duration))`, and `within_interval(T1, StartTime, Duration)`.

6. There are no rules applicable for these and so we evaluate them. We query the sensors for the readings and check the readings against the constraints. We assume that there are built-in predicates to compute "`>`" and `within_interval/3`.

If all the constraints are satisfied, then E would be recognized (by a system Γ, say) to be in the situation represented by the program S; we denote such a relation by $S \Vdash E$ (which by definition holds exactly when the goal S`*>`E succeeds according to the procedure above). Also, note that $S \Vdash E$ whenever $s \in [\![S]\!]_\Gamma$, where s is the perceived situation as perceived by Γ. The definitions provide an account, within our framework, of what it means by the sentence "the system Γ recognizes that E is in situation S."

3 LogicCAP: Embedding Situation Programs in Prolog

Two common questions reasoning with situations attempts to answer are: (1) Given an entity, a set of known situations, and contextual information about the entity obtained from sensors, determine which situation(s) (of those which are known, if any) the entity is in. (2) Given an entity, a situation, and contextual information about the entity obtained from sensors, determine if the entity is in that situation. A solution for (2) can be used in (1) by iterating over the set of situations. Once the situation of the entity is inferred (or computed), situation-actions rules can be applied and actions selected for the system.

We contend that logic programming is generally useful for such reasoning and for supporting the rule-based programming paradigm in context-aware applications. Moreover, as also proposed by Henricksen (2003), situations (there represented as predicates) are a useful programming abstraction for context-aware applications.

In this section, we show how situation programs can be manipulated as first-class entities within a programming language. We embed goals of the

form S*>E into a popular rule-based language such as Prolog so that rule-based context-aware applications can be written. Effectively, we reason with situations by meta-reasoning with situation programs in LogicCAP.

3.1 Syntax

A LogicCAP program comprises rules of the form:

$$\mathcal{A} \text{ :- } \mathcal{G}$$

where ":-" is "if", $\mathcal{G} ::= \mathcal{A} \mid (\mathcal{G}, \mathcal{G}) \mid S$*>$E$, $\mathcal{A}$ is an atomic goal formula, "," denotes conjunction, S is a situation identifier and E is an entity identifier. Rules are facts when $\mathcal{A}$ is true.

3.2 Operational Semantics

We give the operational semantics of LogicCAP as an extension of that of pure Prolog as follows which defines the relation $\vdash$, where P is a LogicCAP program.

$$[true]\frac{}{P \vdash_{\epsilon} \texttt{true}}$$

$$[atom]\frac{P \vdash_{\theta} H \text{:-} G \quad \wedge \quad \gamma = mgu(A, H\theta) \quad \wedge \quad P \vdash_{\delta} G\theta\gamma}{P \vdash_{\theta\gamma\delta} A}$$

$$[in-situation]\frac{ground(S) \quad \wedge \quad ground(E) \quad \wedge \quad S \Vdash E}{P \vdash_{\epsilon} S\text{*>}E}$$

$$[conjunction]\frac{P \vdash_{\theta} G_1 \quad \wedge \quad P \vdash_{\gamma} G_2\theta}{P \vdash_{\theta\gamma} G_1, G_2}$$

The rule for in-situation maps the goal to another evaluation procedure. Hence, LogicCAP rules use backward chaining like Prolog but then utilizes forward chaining in determining situations, i.e. a mix of backward and forward chaining is used in evaluating LogicCAP programs. Note that in this rule, we require that E and S be ground before being used in establishing $\Vdash$. We review this requirement later.

3.3 Programming Idioms and Examples

We show several examples of LogicCAP programs in this subsection. For an application, the programmer needs to develop a (or reuse an existing) collection of situation programs and at least one LogiCAP program.

3.3.1 Reasoning with Situations

We can declaratively and concisely reason over situations in LogicCAP. Because situations are represented as situation programs, we perform metalevel reasoning over situation programs.

Determining actions. We can write rules to select the right actions based on the current situation of an entity. For example, if were writing the logic for a context-aware mobile phone. We could utilize the following rule that says the phone should be in quiet mode if the user is in the meeting and so on.

```
required_phone_mode(quiet) :-
  my_current_user(E), in_meeting_now*>E.
required_phone_mode(quiet) :-
  my_current_user(E), in_lecture*>E.
required_phone_mode(noisy) :-
  my_current_user(E), restaurant*>E.
  :
required_phone_action(change(FromMode,ToMode)) :-
  current_phone_mode(FromMode),
  required_phone_mode(ToMode),
  FromMode \= ToMode.
```

A query to `required_phone_action/1` is used to determine what action (if any) is required for the phone. We assume that there are built-in predicates to return the current user of the phone and predicates to tell what mode the phone is currently in.

Determining situations. We can write rules to search over situations to determine what situation the entity is currently in. For example, the following rule takes a list of possible situations and returns the first situation that the entity matches with.

```
determine_situation(PossibleSituations,E,S) :-
  member(S, PossibleSituations), S*>E.
```

One could also write rules to return all possible situations the entity is in. Given compatibility constraints between situations that says when situations can occur simultaneously or not, one can then determine the conjunction of situations the entity is currently in.

```
can_be_concurrent(eating, driving).
can_be_concurrent(walking, talking).
 :
determine_situations(PossibleSituations,C,E,L) :-
  setof(S, ( member(S, PossibleSituations),
     can_occur_concurrently(C,S), S*>E ), L),
```

The rule determines all situations that the entity is in which are compatible with a current situation `C`. We can also write a rule such as the following that captures selecting the most likely situation out of those which match - which is akin to abductive reasoning in selecting the best explanation.

```
most_likely_situation(PossibleSituations,E,M) :-
  setof(S, ( member(S, PossibleSituations), S*>E ), L),
  most_likely(L,M). % select the most likely situation from the list L
```

`most_likely/2` can be application-specific. One could extend `most_likely/2` to return all sets of compatible situations (from `L`).

We can also write rules to query who out of a given set of entities is in a given situation:

```
sleeping_now(Es,E) :-
  member(E,Es), sleeping*>E.
```

We can also reason about the situations of several users. For example, this rule states that a given message should only be displayed when John and Mary are both in the Chadstone shopping center.

```
display_msg :-
  in_chadstone*>john, in_chadstone*>mary.
```

Relations on situations. Given a collection of identified situations, already represented as situation programs, besides `incompatible/2`, we can define other relationships between situations such as the relation `can_be_concurrent/2`. Other kinds of relationships include composite (or more complex) situations and sub-situations.

```
sleep_walking(sleeping,walking,E) :-
  sleeping*>E, walking*>E.

busy(in_a_meeting, sleeping, on_the_phone, E) :-
    in_a_meeting*>E
  ; sleeping*>E              % using Prolog's 'or'
  ; on_the_phone*>E.

sub_situation_of(R,S,E) :-
  (R*>E)->(S*>E).            % Prolog built-in conditional '->'
```

More complex combinations of situations involving conjunction and disjunction can be defined at the LogicCAP (or meta-) level. The `sub_situation_of/3` predicate can be used to verify if the definitions for `R` and `S` were correct. For example, if situation programs have been written for these two situations, and to the programmer, one is a sub-situation of the other, then one could represent this in a rule as above and verify if that was true for a given entity. Such relationships between situations can be used to define a lattice of situations from a given knowledge base of situation programs.

3.3.2 Situation-Driven Behaviour

So far, in the above programs, we perform backward-chaining on LogicCAP rules, and then forward-chaining with situation programs. The former follows ordinary Prolog execution behaviour and the latter is due to the way we write the rules - with situations implying (causing and/or explaining) the expected observations in sensor readings. We can use such LogicCAP programs to develop a system that provides situation-driven behaviour by inspecting the appropriate sensors frequently.

Suppose we want to develop a system that sends out drink advertisements to a mobile device belonging to a user John when appropriate. For example, consider the following example.

```
display_drink_ad :-
 likely_to_need_drink(john),
 show_drink_ad.
likely_to_need_drink(User) :-
 in_open_air_shopping_mall*>User,in_hot_day*>User.
```

For a user John, this might involve periodically firing a goal such as `display_drink_ad/0` which continually queries the system (and subsequently taking sensor readings) to see if John is likely to need a drink, and only if so, downloads and renders the advertisement. The frequency in which we execute such a goal can be set. The following uses Prolog backtracking and the `repeat/0` predicate which always succeeds on backtracking to continually evaluate `display_drink_ad/0` until it eventually succeeds. The result is effectively polling the appropriate sensors once every second.

```
loop_until_succeed :-
  repeat, sleep(1), % delay 1 second
  display_drink_ad, !.
```

Alternatively, it is useful to have events drive the system. Simply forward-chaining over the rules cannot provide the appropriate behaviour. For example, the goal to retrieve the advertisement should not be carried out until John is detected to likely need a drink. The problem is that both conditions and actions are specified as premises in a rule and are syntactically indistinguishable. As done by Henricksen (2003), and in active databases, rules of the form "`on` *event* `if` *cdn* `then` *action*" can be used to a similar effect. In our context, for such rules to work, changes in sensor readings must be detected and then evaluated against the event description to see if the rule should fire. However, in order to detect a change in sensor readings (when John steps into a particular area), the location sensor needs to be read at some frequency. We might recode the rules as follows.

```
on likely_to_need_drink(john) if true
then
  show_drink_ad,
  render(Text).
likely_to_need_drink(User) :-
  in_open_air_shopping_mall*>User,
  in_hot_day*>User.
```

In-situation goals are then treated as high-level events. For example, `in_open_air_shopping_mall*>User` evaluating to true can be read as the occurrence of the event that the user is in shopping mall. How does one compute these high-level events? One technique is to periodically evaluate the goal `likely_to_need_drink(john)`. Various optimizations are possible. One is to suspend execution during the evaluation of situation programs if constraints are not satisfied, and then to wait until constraints are satisfied. A goal such as `in_open_air_shopping_mall*>User` will not return a result until it evaluates to true (or a timeout occurs). This technique is useful when interfacing with sensors supporting the publish-subscribe model. For such sensors, we modify the evaluation of situation programs to wait on results from sensors. For example, consider the goal `location*(john,chadstone)` which determines if John is in Chadstone. This could be computed as a query to the positioning engine or a subscription to the positioning engine to notify when the event occurs. We need only modify the way sensor predicates are computed in our model, i.e. evaluation of a sensor predicate will result in a subscription to the sensor and evaluation will block within the sensor predicate until a notification is received, afterwhich the sensor predicate returns with the results. Hence, our reasoning is forward chaining until a sensor predicate is evaluated wherein execution is suspended until an event (of results being returned from the sensor) occurs. For optimization, multi-threaded querying of sensors based on ideas from parallel logic programming (Gupta *et al.*, 2001) can be employed.

4 Designing Situation Programs and Properties of a Context-Aware System

We view a context-aware system as containing a number of situation programs which it uses to recognize real-world situations. This section considers the question of what makes a good context-aware system and outlines general requirements on situation programs in order to provide useful and effective behaviour of such a system.

We consider relationships between (i) the situations the developer of a context-aware system wants to model and have the system recognize, and (ii) the situation programs built into the system, giving rise to a statement of favourable properties of a context-aware system. For example, a designer

might want the system to be able to recognize situations when they occur and when they do not occur, and to be able to distinguish between two different situations, or to regard two situations as same. Moreover, it would useful to build situation programs which are reusable for modelling different complex situations.

4.1 Soundness and Completeness

As suggested earlier, a context-aware system effectively tries to recognize situations in the real world: the system therefore links the occurrences of situations in the world to the situations programs it contains.

Given the occurrence of a (real-world) situation i (related to an entity E) and a system Γ is subjected to i, we denote by $satisfied(P)$ the case of $P \Vdash E$ holding.

A system designer tries to represent real world situations via situation programs. For example, given a real-world situation i which the designer can individuate, the designer defines a corresponding situation program for i (assuming particular sensors and sensor predicates in the system), denoted by P_i. At this point, the only link between i and P_i is in the mind of the designer. The system needs to correctly make this link.

It would be a correct design of a system, if when i occurs, we have $satisfied(P_i)$. Conversely, whenever the system determines $satisfied(P_i)$, we would like it to be the case that i occurs. We denote these two relationships as $i \Rightarrow P_i$ and $P_i \rightrightarrows i$, respectively.

In general, with respect to a set of n situations (denoted simply by numbers $\{1, \ldots, n\}$), we say that a context-aware system is

- *complete* if it contains a corresponding set of situation programs $\{P_1, \ldots, P_n\}$ such that $\forall i, \quad i \Rightarrow P_i$, which means that all the intended situations are recognized (in terms of at least one situation program, and this should be P_i), and

- the system is *sound* if $\forall i, \quad P_i \rightrightarrows i$, which means that the system does not give false positives - recognizing a situation as occurring when it is not.

Note that it is up to the designer to verify soundness and completeness of a given system, with respect to various situations, and this could be done during experimental testing of the system. The designer has to subject the

system to various situations and see what the system recognizes. Also, in general, it is difficult to be exhaustive since one cannot subject a system to every possible situation (e.g., to test soundness). Moreover, a real world situation is what the user individuates and there is an element of imprecision about what is conceived in the imagination as a situation. The designer might need to modify the definition of situation programs or employ more sensors to establish a degree of soundness and completeness of the system.

4.2 Perceptual Distinguishability

Given the occurrence of a situation i, there could be several situation programs (e.g., P_i and P_j) that are satisfied. This could be a case of the situation programs each describing an aspect of the real situation occurring, and one could interpret this as two (compatible) situations simultaneously occurring, i.e. that i somehow involves j in the real world. Or it could be that the situation programs P_i and P_j represent overlapping parts of the world, or that the programs P_i and P_j are in error especially if i and j are not compatible (e.g., one is the situation of a person dancing and the other is the situation of the person being dead). In the case of incompatible situations, then, some arbitration is required to choose the best explanation (or situation program) for the given sensor readings. We provided an example on how to do this at the meta-level in Section 3. Here, we note that the incompatibility relationship is clearly not reflexive, is symmetric, and is not transitive in general. For example, a person being in an airplane 10000km above ground is incompatible with the person being on the ground, and a person being on the ground is incompatible with the person being detected by a positioning system as being 10000km above ground, but a person being in an airplane 10000km above ground is compatible with the person being detected by a positioning system as being 10000km above ground. However, one could think of a specific set of situations being represented where the transitive relationship does hold - hence, the designer might use the transitivity property depending on the application.

Also, two different (according to a human perspective) situations might cause the same situation program to be satisfied. For example, i and j both caused P_i to be satisfied. It could be that P_i is not discriminating enough (constraints on the sensor readings are too weak); this is true provided that i and j are indeed incompatible situations. For example, consider two situations, one is the situation of a person dancing and the other is the situation

of the person being dead. If the situation program for dancing is satisfied in both situations, that situation program is clearly not discriminating enough and needs to be improved. However, if we have two situations, one is the situation of a person sitting down on a chair and the other is the situation of the person being dead, then the situation program for sitting down on a chair can be satisfied in both cases.

4.3 Perceiving Non-Occurrence of Situations

If a system is sound and complete, it is already to an extent distinguishing between situations - by correctly matching the situation with the situation program; a different situation should match with a different situation program. However, we might want to build the system in such a way that it does not distinguish between two given situations, or it recognizes the mutual exclusivity (or impossibility of co-occurrence) of two situations. We show how to engineer these stronger perceptual powers by having situation programs that enable the system to recognize the non-occurrence of situations.

What does it mean for the system to distinguish between two situations, or when are two situations undistinguished by the system? Given a system with a set of situation programs, one way to define the relation *undistinguished* between two situations i and j (denoted by $UD(i,j)$) is to say that $UD(i,j)$ holds whenever both i and j causes exactly the same programs to be satisfied. But this leads to transitivity of UD and to a paradox.[11] An implication of this for the developer is a lesson about what not to do: a system, in general, should not be made to infer that two situations are undistinguished using a transitivity of UD rule, since this might cause the system to behave in an unintuitive manner.

To provide a more sensible definition of UD which avoids the paradox, we define the *perceptual indistinguishability* of a system, in a similar way to Shoham and del Val (1991). In particular, we provide a definition of the relation *undistinguished* between two situations in such a way as to avoid transitivity.

We first note that if a situation program P_i is not satisfied (e.g., the system fails to detect that a person is in a meeting, where a meeting is represented according to P), it might or might not mean that situation i

[11] The 'heap paradox' - if two piles of sand differring by one grain cannot be distinguished, then no two piles of sand can be distinguished.

is not occurring (e.g., the person might still be in a meeting even though undetected).[12] Hence, we do not assume negation by failure. Thus, for each situation i, we might not only define P_i but also a situation program which represents the situation that i is not occurring (denote this by $NOTi$) - denote this program by P_{NOTi}. By an abuse of notation (we substitute $NOT(NOTi)$ by i in formulas below) since we take the situation where "the situation that i is not occurring" is not occurring as meaning the situation that i is occurring, i.e. we define $P_i = P_{NOT(NOTi)}$. So if i refers to the person being in a meeting, P_{NOTi} refers to a situation program where the person is not in a meeting, and the system has to detect that. To avoid confusing and contradictory results, the designer should strive to define P_i and P_{NOTi} such that always

$$\neg(satisfied(P_i) \wedge satisfied(P_{NOTi}))$$

holds, i.e. it is never the case that both are satisfied. We call this the *consistency condition.* But note that it might be the case where both P_i and P_{NOTi} are not satisfied - P_i is not satisfied does not automatically mean that P_{NOTi} will be - when, for example, neither i nor $NOTi$ are judged to be possible explanations for the sensed information.

Assuming that a system being designed to recognize situations 1 to n has now the situation programs $\{P_1, \ldots, P_n\}$ (and their corresponding complements $\{P_{NOT1}, \ldots, P_{NOTn}\}$). We then define UD as follows based on (Shoham and del Val, 1991):

$$\begin{aligned} UD(i,j) =_{def} & \neg(satisfied(P_i) \wedge satisfied(P_{NOTj})) \\ & \wedge\neg(satisfied(P_j) \wedge satisfied(P_{NOTi})) \end{aligned}$$

In other words, two situations i and j are distinguished by a system if the system's situation programs are such that $i \Rightarrow P_i$ and $i \Rightarrow P_{NOTj}$, i.e., i occurring causes P_i to be satisfied and P_{NOTj} to be satisfied, or $j \Rightarrow P_j$ and $j \Rightarrow P_{NOTi}$, i.e., j occurring causes P_j to be satisfied and P_{NOTi} to be

[12] This is consistent with our view of the situation rules in an abductive manner: given "if situation then observation" and "observation", we infer "situation". But given "if situation then observation" and "not observation", where "not p" means that p is not provable (as different from ¬ p) we do not necessarily infer the non-occurrence of the situation by abduction (since we do not use deduction (modus tollens) here). We assume that the rule captures the idea that the situation is a possible explanation for the given observation, but if that observation is not observed, the non-occurrence of the situation may not necessarily hold.

satisfied. This means that the recognition of one situation should come with the recognition of the non-occurrence of another (rules out the other).

This definition of UD is not transitive (following the proof in (Shoham and del Val, 1991)) for suppose that at a given point in time, the system shows that $\neg satisfied(P_i)$ and $\neg satisfied(P_{NOTi})$ are true, i.e. the system is not able to say anything about a situation i,[13] but is recognizing j (i.e. $satisfied(P_j)$), then $UD(j,i)$ and $UD(i,NOTj)$ are true but $UD(j,NOTj)$ is false. Also, note that reflexivity of UD is preserved provided the consistency condition holds, i.e. $UD(i,i)$ holds for any i, and symmetry holds.

UD relationships might be used as part of the specification of a context-aware system, with implications on what situation programs the system should have. For example, the system which needs to distinguish between the situations `walking` and `running`, and the situations `in_a_meeting` and `available`, can be specified with the statements $\neg UD(\texttt{walking},\texttt{running})$, and $\neg UD(\texttt{in\_a\_meeting},\texttt{available})$. From the definition of UD above, the programs $P_{\texttt{walking}}$, $P_{\texttt{running}}$, $P_{\texttt{NOTwalking}}$, and $P_{\texttt{NOTrunning}}$ need to be defined and in such a way that $\neg UD(\texttt{walking},\texttt{running})$ holds, and similarly, the four programs for `in_a_meeting` and `available`. For example, if the system is build this way, in a run of the system, when we have $satisfied(P_{\texttt{walking}})$, then the system would also conclude $satisfied(P_{\texttt{NOTrunning}})$. This implies that the system (with its sensors and set of situation programs) has adequate perceptual ability to "see" not only the occurrence of a situation but also the non-occurrence of another situation implied by the occurrence of the first. If the system is not able to also conclude $satisfied(P_{\texttt{NOTrunning}})$ or the system does not contain a definition for $P_{\texttt{NOTrunning}}$, then the system is leaving open the possibility that while the person is walking, he or she might also be running (i.e. $satisfied(P_{\texttt{running}})$ is possible) or the person is not running but the system is not detecting it, i.e. the system is unable to recognize the close relationship between `walking` and `running` that the occurrence of one situation always excludes the other. It is one level of perceptual ability for a system to recognize a situation, and another (higher) level of perceptual ability if the system recognizes not only the situation but (concurrently) also recognizes the non-occurrence of another situation (ruled out by the occurrence of the first) - the system is supplying more information in the latter case.

We also note that the above completeness and soundness definitions can

[13] This could be due to a number of reasons including a faulty definition of P_i or P_{NOTi}.

be applied for a system containing not only situation programs but also the corresponding "NOT" situation programs. Given that the real-world is such that i occurring means that j is not occurring, completeness implies that the UD relationships are indeed upheld, since both the occurrence and non-occurrence of situations will be detected.

The ability to distinguish between situations does not help to determine if two situations are compatible or incompatible. If two situations are distinguished, they may still be incompatible or compatible. For example, driving and sitting might be distinguished by the system, but they can both occur at the same time - they are *compatible*. For each pair of incompatible situations i and j, and if needed for an application, we would like the condition that $\neg UD(i, j)$ to be added to the specification of the system. However, if two situations i and j are compatible, it is not sensible to have the system satisfy $\neg UD(i, j)$ since the occurrence of one does not imply the non-occurrence of the other.

Based on Barwise (1989), it would also be useful if a situation i (e.g., a soccer game is on between Brazil and Japan) can reasonably be viewed (by a human) as containing j (e.g., Brazilian players are running around in the field), then whenever $i \Rightarrow P_i$, then $i \Rightarrow P_j$, or that this "containment" can be represented by a rule such as "whenever $satisfied(P_i)$, then $satisfied(P_j)$". However, whether such a rule holds depends on the exact definition of P_i and P_j chosen by the designer. One can define such relationships between situation programs at the meta-level as we showed using `sub_situation_of/3` in Section 3.3.1.

4.4 Modularity

One might wish to add new situation programs and remove existing programs over time. For example, a context-aware system might already have a number of sensors and a set of situation programs. As new situations need to be recognized, new situation programs that utilize existing sensors (and sensor predicates) can be defined.

New sensor predicates which process sensor readings (from existing sensors) into context might be added. Existing sensors can be replaced, possibly without affecting the structure (or signature) of existing sensor predicates, and therefore, leaving situation programs unchanged. If new context information need to be considered, new sensor predicates can be added. And if new sensors or sensor predicates are added affecting the definition of situa-

tion programs, then only the affected situation programs (those with sensor predicates using the new sensors) need to be changed.

Since LogicCAP programs refer to situation programs by their names at the meta-level, situation programs can be modified internally without affecting the LogicCAP rules. For example, in the mobile phone program in Section 3.3.1, the situation program for `in_lecture` might be changed (e.g., a new sensor is added which will improve the certainty of recognizing this situation) without affecting the program.

We have also shown how to represent complex situations in terms of other situations in Section 3.3.1. Situation programs can be built which are reused (as constituents) to recognize different complex situations. For example, as shown earlier, `sleeping` was used both to define `sleeping_walking`, and `busy`.

4.5 Discussion

In summary, the designer should build situation programs (with their corresponding NOT duals) into a context-aware system in such a way that the system will distinguish between situations that it should distinguish and should not distinguish between situations that it should not distinguish - the situation programs are defined such that UD (as defined above) holds under the right circumstances: when two situations i and j are not to be distinguished, then $UD(i, j)$ should hold, and conversely. Moreover, the situation programs should enable the system to respect the containment relation between real-world situations.

A context-aware system should be sound, complete, and have adequate perceptual power, as least for the application intended. The situation programs should also be built with reuse in mind. As mentioned earlier, it is generally difficult to test that a system satisfy such properties since it is not possible to artificially subject a system to all possible situations one can conceive. While our discussion has been theoretical, we have provided a formulation of these properties (within our representation of situation) which are typically not stated explicitly, or merely stated informally in the literature. Also,we envision that limited testing by subjecting the system to a limited set of situations is possible once a system is built - in which case the situation programs might be modified (or even sensors replaced, or added to the system) in order for the system to work correctly.

5 Related Work

Research has begun to look at frameworks for context-aware systems more generally, independently of specific applications, including context middleware, infrastructures and toolkits (e.g., (Dey, 2000; Henricksen, 2003; Biegel and Cahill, 2004; Hong and Landay, 2001)). Such work facilitates the building of context-aware systems. Tools for end-users to program context-aware systems are also being built on top of these infrastructures. For example, iCAP (Sohn and Dey, 2003) allows end-users to program a context-aware application by specifying rules. Indeed, such rule-based programming seems intuitive and well-suited to context-aware applications. We take rule-based programming further with our approach by representing situations and not only programming rule-based triggers for context-aware actions.

There has been work by Ranganathan and Campbell (2003) and Katsiri and Mycroft (2003) where first order logic is used for representing and reasoning with context, and work in (Henricksen, 2003) where first-order logic is used to describe and reason with situations. However, they do not use a modular approach or meta-reasoning as we do in LogicCAP. Moreover, they do not consider an abductive semantics as we do for our situation rules.

As mentioned in the introduction, related is work on using ontologies and the Semantic Web for representing and reasoning with context (e.g., (Chen *et al.*, 2003)). We have so far not utilized ontologies, but feel our work is complementary to such work. For example, we can use ontologies to provide a vocabulary for situation predicates in our situation programs, so that such programs might be shared across different systems (or even retrieved from a store over the Web). Also, ontologies can provide vocabularies and additional semantics to sensor predicates - such predicates can represent context attributes specified by an ontology. Standard signatures for sensor predicates facilitates reuse of situation programs. Moreover, we acknowledge that there could be many different ways (different sensors and different context attributes used) for recognizing a situation. To quote Woods (1978) long before the advent of pervasive computing: "...there are in general many different ways in which the current situation might be described, and it is not clear how one should construct such a description." For example, to recognize if there is a meeting going on, there could be several approaches (as can be viewed from the current literature). One approach is to use weight sensors on the floor. Another approach might be to track the location of participants. A third approach is to track the co-location of filled coffee cups in a room

(Gellersen *et al.*, 2002). A fourth approach might be to combine location and time, characteristics of devices in the room (e.g., powerpoint is being used, light is on, etc) and personal diaries and schedules of participants (Chen *et al.*, 2004a; Ranganathan and Campbell, 2003). The actual approach employed might depend on the sensors available and the infrastructure at hand, and other pragmatic considerations. Such approaches can be combined but an ontology can be useful for describing the situation of a meeting using a common vocabulary of concepts, or to represent that the different notions of meeting do coincide (e.g., all the different approaches could refer to the concept of meeting in the same ontology).

Tazari (2003) gives a characterization of situation as ranges on sensor readings as follows. Given a situation s, let $k_1, \ldots, k_n$ be a set of context attributes (e.g., location is one attribute, and temperature might be another), $v(k_i, t)$ denotes the value of k_i at time t, and V_i denotes the range of possible values that are acceptable for k_i within s. Then, s is recognized by the following characteristic function which states that a situation is recognized when the values of the context attributes are all within a certain range: $\chi_s : Time \rightarrow \{true, false\}$, defined as $\chi_s(t) = \bigwedge_{i=1}^{n}(v(k_i, t) \in V_i)$. Our characterization can be viewed as a generalization of this approach; instead of constraints on sensor readings (values returned by sensors for context attributes) in the form of the sets V_is, we represent the constraints using relations and variables in logic programming rules.

There has been other work on situation recognition, dating back a decade and recently (Dousson *et al.*, 1993; Mouthaan *et al.*, 2003). Such work tend to model situations as event patterns. Situation recognition boils down to analysing real-time streams of sensor data (which might be abstracted into events) and matching them with stored event patterns. Our approach models situations differently, via situation programs, and also enable (meta-)reasoning with situations within a declarative programming language. The notion of events can be captured within sensor predicates as noted in Section 3.3.2.

Pinheiro (2002) discusses the use of situation theory to describe real-world scenarios in a more formal way. A scenario description comprises a title identifying the scenario, the scenario's goal, the scenario's context (i.e. the scenario's initial state, pre-conditions, and the space-time location), resources, actors, and episodes (or actions of actors involving resources). Goals and contexts are described by situations, where a situation is described by a set of infons (each infon a relationship among individuals). There are

notions of inclusion, consistency, and part-of between situations within the framework. They do not consider the notion of sensors in their scenario representation (as their work is not tailored for context-aware computing), and do not make connections with logic programming or meta-programming as we do.

We have previously considered connnections with context-aware Web applications in (Loke, 2004). There, we introduced the idea of situation programs and LogicCAP, but did not discuss their denotational semantics or properties of context-aware systems and emphasized integration with the Web.

6 Conclusion and Future Work

We have presented a declarative approach to building context-aware pervasive systems. We introduced the notion of the situation program which highlights the primacy of the situation program for building context-aware systems. We provided semantics for situation programs and discussed how the design of situation programs can affect the properties of a context-aware system. The situation programs need to be defined in such a way as to provide favourable properties to the system. Such properties provide a general guide as to design aspirations for situation programs.

We have also shown how to manipulate situation programs within LogicCAP, and illustrated the convenience of meta-reasoning with situations within LogicCAP. The "in-situation" operator captures within the language a common form of reasoning in context-aware applications, which is to ask if an entity is in a given situation. The computation model for LogicCAP has been backward-chaining on LogicCAP rules (following ordinary Prolog execution behaviour), and then forward-chaining with situation programs due to the way we write the rules, with situations implying (causing/explaining) the expected observations in sensor readings.

Three key benefits of our approach are as follows:

1. The approach encourages representing and reasoning with situations using a declarative language, providing a high-level of abstraction.

2. The approach supports building context-aware systems incrementally by providing modularity and separation of concerns.

3. The approach is amenable to formal analysis based on logic programming theory.

There are several avenues for further work. While we can reason with disjunctions of situations at the meta-level, we are investigating disjunctive constraints within situation programs. It would be convenient to be able to use "or" between goals in the rules. We have only dealt with in-situation goals where the operands of $S \Vdash E$ are grounded. Queries where S and E are variables can be considered, to find all entities in a given situation or all situations for a given entity's context. Incorporating reasoning with uncertainty can also be investigated. We are continuing to prototype more situation programs and a complete context-aware system on the basis provided in this paper. Practical experiences will need to be gained. Since situation programs are logic programs, it would also be interesting to consider what new situation programs are formed if we composed these logic programs according to operators similar to that described by Brogi (1993). Lattices of situations might be constructed as lattices of situation programs which can help the system understand and reason with relationships between situations as noted by Woods (1978).

References

V. Akman. Context in Artificial Intelligence: A Fleeting Overview (English version of Contesti in intelligenza artificiale: una fugace rassegna). In C. Penco, editor, *La Svolta Contestuale*. McGraw-Hill, Milano, 2002.

L. Barkhuus. Context Information vs. Sensor Information: A Model for Categorizing Context in Context-Aware Mobile Computing. In *Symposium on Collaborative Technologies and Systems*, pages 127–133, San Diego, CA, 2003.

J. Barwise. *The Situation in Logic*. CSLI, 1989.

J. Barwise, J.M. Gawron, G. Plotkin, and S. Tutiya, editors. *Situation Theory and its Applications*. CSLI, 1991.

J. Barwise and J. Perry. *Situations and Attitudes*. Cambridge: MIT-Bradford, 1983.

G. Biegel and V. Cahill. A Framework for Developing Mobile, Context-Aware Applications. In *2nd IEEE Conf. on Per-*

vasive Computing and Communications, March 2004. Available at http://www.dsg.cs.tcd.ie/~biegelg/research/publications/TCD-CS-2004-04.pdf.

P. Bouquet, C. Ghidini, F. Giunchiglia, and E. Blanzieri. Theories and Uses of Context in Knowledge Representation and Reasoning. *Journal of Pragmatics*, 35(3), 2003.

P. Brezillon. Representation of Procedures and Practices in Contextual Graphs. *The Knowledge Engineering Review*, 18(2):147–174, 2003.

A. Brogi. *Program Construction in Computational Logic*. PhD thesis, University of Pisa, 1993.

H. Chen, T. Finin, and A. Joshi. An Ontology for Context-Aware Pervasive Computing Environments. In *Workshop on Ontologies and Distributed Systems, IJCAI-2003*, August 2003.

H. Chen, T. Finin, and A. Joshi. A Context Broker for Building Smart Meeting Rooms. In Craig Schlenoff and Michael Uschold editors, *Proceedings of the Knowledge Representation and Ontology for Autonomous Systems Symposium, 2004 AAAI Spring Symposium*, pages 53–60, Stanford, California, 2004. AAAI Press, Menlo Park, CA.

H. Chen, T. Finin, and A. Joshi. An Ontology for Context-Aware Pervasive Computing Environments. *The Knowledge Engineering Review (special issue on "Ontologies for Distributed Systems")*, 18(3):197–207, 2004.

K.J. Devlin. Situations as Mathematical Abstractions. In J. Barwise, J.M. Gawron, G. Plotkin, and S. Tutiya, editors, *Situation Theory and its Applications*. CSLI, 1991.

A.K. Dey. *Providing Architecture Support for Building Context-Aware Applications*. PhD thesis, Georgia Institute of Technology, 2000.

A.K. Dey. Understanding and Using Context. *Personal and Ubiquitous Computing Journal*, 5(1):5–7, 2001.

C. Dousson, P. Gaborit, and M. Ghallab. Situation Recognition: Representation and Algorithms. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, Chambry, France, pages 166–172, 1993.

H-W. Gellersen, A. Schmidt, and M. Beigl. Multi-Sensor Context-Awareness in Mobile Devices and Smart Artifacts. *Mobile Networks and Applications (MONET)*, 7(5):341-351, October 2002.

G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, July 2001.

R. Headon. Movement Awareness for a Sentient Environment. In *Proceedings of the 1st Conference on Pervasive Computing and Communications (PerCom 2003)*. IEEE Computer Society Press, March 2003.

K. Henricksen. *A Framework for Context-Aware Pervasive Computing Applications*. PhD thesis, University of Queensland, 2003.

K. Henricksen, J. Indulska, and A. Rakotonirainy. Modeling Context Information in Pervasive Computing Systems. In *Proceedings of the Workshop on Context Modelling and Reasoning (CoMoRea'04) at the 2nd IEEE International Conference on Pervasive Computing and Communications*, pages 33–37, March 2004.

J.I. Hong and J.A. Landay. An Infrastructure Approach to Context-Aware Computing. *Human-Computer Interaction*, 16, 2001. Available at http://guir.berkeley.edu/projects/confab/pubs/context-essay-final.pdf.

A. Hopper. The Royal Society Clifford Paterson Lecture - Sentient Computing. *Phil. Trans. R. Soc. Lond*, 358:2349–2358, 2000. Available at http://www-lce.eng.cam.ac.uk/publications/files/tr.1999.12.pdf.

E. Katsiri and A. Mycroft. Knowledge Representation and Scalable Abstract Reasoning for Sentient Computing Using First-Order Logic. In *Proc. Challenges and Novel Applications for Automated Reasoning (CADE-19 Workshop)*, July 2003. Available at http://www.cl.cam.ac.uk/users/am/papers/nads03.pdf.

K. Koile, K. Tollmar, D. Demirdjian, H. Shrobe, and T. Darrell. Activity Zones for Context-Aware Computing. In *Proceedings of the 5th International Conference on Ubiquitous Computing (UBICOMP 2003)*, 2003.

O. Lehmann, M. Bauer, C. Becker, and D. Nicklas. From Home to World - Supporting Context-Aware Applications through World Models. In *Proceedings of the 2nd IEEE Annual Conference on Pervasive Computing and Communications (PERCOM'04)*. IEEE Computer Society, 2004.

J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

S.W. Loke. Logic Programming for Context-Aware Pervasive Computing: Language Support, Characterizing Situations, and Integration with the Web . In *Proceedings of the IEEE/WIC/ACM Conference on Web Intelligence (WI'04)*, 2004. (to appear)

C. Matheus, M. Kokar, and K. Baclawski. A Core Ontology for Situation Awareness. In *Proceedings of FUSION03*, pages 545–552, July 2003. Available at http://vistology.com/papers/FUSION03.pdf.

J. McCarthy. Notes on Formalizing Contexts. In Ruzena Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 555–560, San Mateo, California, 1993. Morgan Kaufmann.

R.E. McGrath, A. Ranganathan, R.H. Campbell, and M.D. Mickunas. Use of Ontologies in Pervasive Computing Environments. Technical report, UIUCDCS-R-2003-2332 ULU-ENG-2003-1719, April 2003. Available at http://mummy.intranet.gr/includes/docs/MUMMY-D11y1-ZGDV-CtxtAwr-v02.pdf.

T. Moran and P. Dourish, editors. *Context-Aware Computing*. Lawrence Erlbaum Assoc Inc, 2002.

Q.M. Mouthaan, P.A.M. Ehlert, and L.J.M. Rothkrantz. Situation Recognition as a Step to an Intelligent Situation-Aware Crew Assistant System. In *Proceedings of the 15th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2003)*, Nijmegen, The Netherlands, pages 219–226, October 2003.

M. Muhlenbrock, O. Brdiczka, and J-L. Meunier D. Snowdon. Learning to Detect User Activity and Availability from a Variety of Sensor Data. In *Proceedings of the 2nd IEEE Annual Conference on Pervasive Computing and Communications (PERCOM'04)*. IEEE Computer Society, 2004.

D.J. Patterson, L. Liao, D. Fox, and H. Kautz. Inferring High-Level Behaviour from Low-Level Sensors. In *Proceedings of the 5th Annual Conference on Ubiquitous Computing (UBICOMP 2003)*, 2003.

S. Peters and H.E. Shrobe. Using Semantic Networks for Knowledge Representation in an Intelligent Environment. In *Proceedings of the 1st IEEE Annual Conference on Pervasive Computing and Communications (PERCOM'03)*, pages 323–329. IEEE Computer Society, 2003.

R.W. Picard. *Affective Computing*. MIT Press, 1997.

F. Pinheiro. Preliminary Thoughts on Using Situation Theory for Scenario Modelling. In *Workshop on IDEAS 2002*, La Habana, Cuba, 2002.

A. Ranganathan and R.H. Campbell. An Infrastructure for Context-Awareness Based on First Order Logic. *Personal and Ubiquitous Computing Journal*, 7:353–364, 2003.

B.N. Schilit, N.I. Adams, and R. Want. Context-Aware Computing Applications. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 85–90. IEEE Computer Society, December 1994.

Y. Shoham and A. del Val. A Logic for Perception and Belief. Technical Report STAN-CS-91-1391, Dept. of Computer Science, Stanford University, September 1991. Available at ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/91/1391/CS-TR-91-1391.pdf.

T. Sohn and A.K. Dey. iCAP: An Informal Tool for Interactive Prototyping of Context-Aware Applications. In *Interactive Poster in the ACM Conf. on Human Factors in Computing Systems*, pages 974–975, April 2003.

E.M. Tapia, S.S. Intille, and K. Larson. Activity Recognition in the Home Using Simple and Ubiquitous Sensors. In A., editor, *Pervasive 2004, LNCS 3001*, pages 158–175. Springer-Verlag, 2004.

S. Tazari. Context-Awareness and Knowledge Representation D11. Technical report, MUMMY (IST-2001-37365), 2003. Available at http://www.cs.uiuc.edu/Dienst/UI/2.0/Describe/ncstrl.uiuc_cs/UIUCDCS-R-2003-2332.

E. Tin and V. Akman. Computational Situation Theory. *SIGART Bulletin*, 5(4):4–17, 1994.

X.H. Wang, T. Gu, D.Q. Zhang, and H.K. Pung. Ontology Based Context Modeling and Reasoning using OWL. In *Proceedings of the Workshop on Context Modelling and Reasoning (CoMoRea'04) at the 2nd IEEE International Conference on Pervasive Computing and Communications*. IEEE Computer Society Press, March 2004.

W.A. Woods. Taxonomic Lattice Structures for Situation Recognition In *Proceedings of the Theoretical Issues in Natural Language Processing*, Urbana-Campaign, Illinois, United States, pages 33 – 41, 1978.

B. Yoshimi. On Sensor Frameworks for Pervasive Systems. In *Proceedings of the Workshop on Software Engineering for Wearable and Pervasive Computing at the 22nd International Conference on Software Engineering*, 2000.