A Context-Driven Programming Model for Pervasive Spaces

Hen-I Yang, Jeffrey King, Abdelsalam (Sumi) Helal, Erwin Jansen

Pervasive and Mobile Computing Lab, University of Florida
{hyang, jck, helal, ejansen}@cise.ufl.edu
www.icta.ufl.edu

Abstract. This paper defines a new, context-driven programming model for pervasive spaces. Existing models are prone to conflict, as it is hard to predict the outcome of interleaved actions from different services, or even to detect that a particular device is receiving conflicting instructions. Nor is there an easy way to identify unsafe contexts and the emergency remedy actions, or for programmers and users to grasp the complete status of the space. The programming model proposed here resolves these problems by improving coordination by explicitly defining the behaviors via context, and providing enhanced safety guarantees as well as a real-time, at-a-glance snapshot of the space's status. We present this model by first revisiting the definitions of the three basic entities (sensors, actuators and users) and then deriving at the definition of the operational semantics of a pervasive space and its context. A scenario is provided to demonstrate both how programmers use this model as well as the advantages of the model over other approaches.

1 Introduction

As the field of pervasive computing matures, we are seeing a great increase in the number and the variety of services running in a pervasive space. Ad-hoc approaches to developing these spaces become unwieldy with this complexity. As other researchers have found, a programming model for pervasive spaces is required, so that these environments can be assembled as software. The keystone of such a model is context, because of the dynamic and heterogeneous nature of these environments. Several systems and tools have been created to support context-aware development of pervasive spaces [1, 2, 3, 4].

While these existing programming models provide a convenient means for specifying the rules and behaviors of the applications with a formulated and systematic process, they do not provide an easy way to grasp the overall state of the smart space with a glance. In these models, obtaining the "big picture" of the space requires intentional effort; programmers first must know exactly what to look for, then explicitly query the system for particular data, and then compose and interpret the collected data, weaving them into an overall understanding.

As the number of applications grows, or the scenarios become more complicated, or services are provided by different vendors, conflict is almost certain. The existing

models provide little support to detect conflicts. Contradictory commands may be issued to the same actuator by different applications, which can put one or more of the applications into an erroneous state, or can even damage the targeted device. Nor do these models provide any guarantee about the safety of interleaving commands from different applications. In these service-oriented models, each service is focused on its own goals, ignoring the potential side-effects of concurrent access to shared resources – side-effects that could put the space into a dangerous or anomalous state.

These existing models also lack a simple way to identify, at a global level, unsafe contexts that should be avoided and the means to exit them should they arise. Instead, each service must have its own routines for detecting and handling these situations. Not only does this place an unrealistic demand on services developers, but safety guarantees in the space are infeasible, as a single missing or erroneous handler in a single service could allow the space to be stuck in one of these impermissible contexts.

Since our work in smart spaces has focused on creating assistive environments for seniors or people with special needs [5], the safety and reliability aspects of the programming model are especially critical. During the creation of our two smart homes – a 500 sq. ft. in-lab prototype, and the full-scale, 2500 sq. ft. freestanding Gator Tech Smart House [6] – we found that the existing models and tools could not provide us with satisfactory safety guarantees and conflict management capability. We began expanding the ideas of context-oriented design to create a new programming model for pervasive spaces.

The main idea of our approach is to employ standard ontology to build a context graph that represents all possible states of interest in a smart space. Contexts in the graph are marked as desirable, transitional, or impermissible, and the goal is to take actions to always lead the smart space to desirable contexts, and to avoid impermissible contexts. At run time, the torrent of raw readings from sensors are interpreted and classified into active contexts, and the associated actions are set in motion to drive towards and strive to maintain desirable contexts.

In other models, knowledge about the behavior of the smart space, such as the actions, conditions and other application logic that constitute services, are all hidden in the source code, and programmers can only make educated guesses regarding the exact sequence of actions from the name of the methods provided. Our model, however, uses standard ontology and specifies the behavior explicitly using active contexts as the conditions and explicit commands to be issued to various actuators based on the current active contexts. This explicitness improves the coordination and mitigates conflicts between devices, users and services, especially in highly heterogeneous and dynamic environments such as smart spaces. It also guides the programmers to identify illogical or erroneous actions at compile time, which can greatly boost the safety of programming pervasive spaces.

Since services provided in a smart space using our model are programmed as various actions to be taken based on active contexts, it is easy to prevent and to verify no conflicting actions are associated with each context. Decisions on action plans based on the active contexts also avoid the danger that can be caused by executing interleaved actions issued by different services. The explicit designation of impermissible contexts allows the system to take emergency actions and exit the

dangerous situations. All of these contribute to the enhanced safety guarantees offered by our model.

Finally, the explicit context graph and the visualization of currently active contexts allow real-time, at-a-glance snapshots of the space's status, which give the users and programmers alike an understanding of the overall picture of the smart space.

We continue in section 2 of the paper with an overview of related research in programming models for pervasive spaces. In sections 3 through 5, we describe the components of the model, providing formal definitions for devices, users and their interactions in smart spaces. A scenario demonstrating the advantages and usage (including programming procedures) of our context-driven programming model is provided in section 6. Finally, we conclude in section 7 with a summary of the practicality and advantages of our model.

2 Related Work

The majority of ubiquitous computing research involving implemented systems has been pilot projects to demonstrate that pervasive computing is useable [7]. In general, these pilot projects represent ad-hoc, specialized solutions that are not easy to replicate. However, one thing that these applications do share is the notion of context.

Context-aware computing is a paradigm in which applications can discover and take advantage of contextual information. This could be temperature, location of the user, activity of the user, etc. 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 application themselves [8].

In order to ease the development of pervasive applications, effort has been placed into developing solutions that enable easy use of context. There are two main approaches: libraries and infrastructure. A library is a generalized set of related algorithms whereas an infrastructure is a well-established, pervasive, reliable, and publicly accessible set of technologies that act as a foundation for other systems. For a comparison between the two, see [9].

The Context Toolkit [1, 10] provides a set of Java objects that address the distinction between context and user input. The context consists of three abstractions: widgets, aggregators and interpreters. Context widgets encapsulate information about a single piece of context, aggregators combine a set of widgets together to provide higher-level widgets, and interpreters interpret both of these.

In the Gaia project, an application model known as MPACC is proposed for programming pervasive space [11], and includes five components with distinct critical functionalities in pervasive computing applications. The model provides the specification of operation interfaces, the presentation dictates the output and presentations, the adaptor converts between data formats, the controller specifies the rules and application logic and the coordinator manages the configurations.

EQUIP Component Toolkit (ECT) [12], part of the Equator project, takes an approach more similar to distributed databases. Representing entities in the smart space with components annotated with name-value pair property and connection, ECT

provides a convenient way to specify conditions and actions of the applications, as well as strong support in authoring tools, such as graphic editors, capability browsers and scripting capabilities.

The use of ontology in our model is related to the notion of Semantic Web services [13]. The idea is that by giving a description of a web service we can automate tasks such as discovery, invocation, composition and interoperation. The Semantic Web makes use of description logic to describe a service. Description logics are knowledge representation languages tailored for expressing knowledge about concepts and concept hierarchies. An agent can use this description to reason about the behavior or effect of the invocation of a web service.

The SOCAM architecture [2] is a middleware layer that makes use of ontology and predicates. There is an ontology describing the domain of interest. By making use of rule-based reasoning we can then create a set of rules to infer the status of an entity of interest. The SOCAM research shows that ontology can be used as the basis of reasoning engines, but such engines are not suitable for smaller devices due to the computational power required.

CoBrA [3] is a broker-centric agent architecture. At the core of the architecture is a context broker that builds and updates a shared context model that is made available to appropriate agents and services.

Our programming model differs from the models and tools summarized in this section in that we give a formalization of the pervasive space and solely make use of description logic to interpret the current state of the world. This guarantees that contexts declared can actually occur in the space and prevents us from simultaneously activating contradictory contexts. We also explicitly describe the effect of entities in a smart space, allowing the system to detect conflicting or dangerous behaviors at compile time. This provides safety assurance in the pervasive space at all times.

3 The Physical World

Unlike traditional computing systems, which primarily manipulate a virtual environment, smart spaces deal with the physical world. We observe and interact with the world. We consider the world to be in a certain state at a given time. Smart spaces encode the physical world using three entities: sensors, actuators, and users.

Sensors and actuators are active objects in the space. Sensors provide information about a particular domain, giving the system information about the current state of the space. A sensor cannot change the state of the space – it can only observe. Actuators are devices that influence the state of the space. They can influence the state because the invocation of an actuator has at least one intentional effect on a particular domain. For example, an intentional effect of the air-conditioner actuator is to cool the room, affecting the “temperature” domain. This change can be observed by a temperature-domain sensor, such as a thermometer.

Objects other than sensors and actuators are “passive” or “dumb” objects, and cannot be queried or controlled by the space. These entities, such as sofas or desks, are therefore irrelevant to the programming model. They may be embedded with

active objects (e.g., a pressure sensor in the seat of a couch), but it is the sensors or actuators that are important, not the passive object itself.

Users are special entities in the space. While they are of course not "wired" into the system like sensors or actuators, users are an integral part of the smart space, as it is their various and changeable preferences and desires that drive the operation of the space.

3.1 Observing the World with Sensors

We consider our world of interest to be $U = \prod Dj$ where Dj is a domain, within the bounds of the space, that is observable by sensors. We will use $u \in U$ to denote aspects of the current state of the world.

Sensors provide data about the current state of affairs. A smart space typically contains many physical sensors that produce a constant stream of output in various domains. As this data is consumed by the smart space, each physical sensor can be treated as a function the space can call to obtain a value.

Definition 1 (Sensor). A sensor is a function that produces an output at a given time in a particular domain.

$$f_i^j : U \rightarrow Di$$

Notice that we can have multiple sensors observing a single domain *Di*. This is indicated by f_i^1, f_i^2, etc. Sensors in the same domain may produce different values due to different locations, malfunction or calibration issues. The smart space system will be responsible for correctly interpreting data. We can group all available sensors together and define $f = \prod f_i$ to be a snapshot of all sensors at a point in time.

3.2 Influencing the World with Actuators

Actuators allow the smart space to affect the physical world. We model the actuators as a set A with elements $a_i \in A$. Our model assumes that while an actuator is turned off it has no effect on the world. In this sense, an actuator that is off is the same as an actuator that does not exist in the space.

Every actuator in the space has certain intentional effects, which are the goals we intend to achieve with activation of the actuator. Programmers specify the intentional effect by defining how activation of the actuator will affect the state of the pervasive space – in other words, given the current state, to what new state will the space transition when the actuator is turned on. We formalize the intentional effect of an actuator as follows:

Definition 2 (Intentional Effect). An intentional effect of an actuator *a* is the set of states that possibly can arise due to invocation of that actuator:

$$g_a : U \rightarrow 2^U$$

An actuator may have more than one intentional effect. For example, turning on a vent can be used both to improve the air quality and raise the overall power draw.

4 Inhabiting the World as a User

So far we have described a space consisting of sensors and actuators. The key element that is missing is the user. To model the user we roughly follow the belief-desire-intention [14] model, in the sense that the user observes the state of the space and, based upon this information, commits to a plan of action. We have the following ingredients:

- **Desires:** A user has a set of preferences about the state of the world. These preferences dictate how the world should be at that given moment. For example, when a user is going to bed, he or she would like the doors locked. When the state of the space changes, the user's preference can change as well.
- **Belief:** This is the current observed state of the world, interpreted by taxonomy. Ideally the system's perception of the state of the world should the user's perception. For example, if the user considers 75 - 80 ˚F as warm, then the system should activate the *warm* context at that temperature range.
- **Intention:** This is the action plan to which the user commits. In a smart space this would be the activation of a set of actuators. The aim of the activation is to fulfill the desire as previously mentioned.

The idea is that the user observes that state of the world. The user then classifies the state according to the users' specific taxonomy. This classification gives rise to a set of contexts that are active, and each context is associated with a set of behaviors. The intention of these behaviors is to change the current state to a more desirable state. For example, if user interprets the world as *Cold Indoors,* we turn on the heater with the intention to reach the desired context of *Warm Indoors*.

5 The Context-Driven Programming Model for Pervasive Spaces

Our smart space programming model involves interpreting the sensor readings, controlling the actuators and capturing and reasoning about the users' desires.

5.1 Interpreting Sensor Data

Dealing directly with sensor data is rather awkward. Instead, we would like to work with higher level information that describes the state of the world. A natural choice for describing knowledge is to use description logics. Using description logic we can define an ontology that describes the smart space. We will restrict ourselves to taxonomic structures, and therefore will have no need to define roles. The language *L* we define here is similar to ALC without roles. This language, and description logics

in general, are described in more detail in [15]. We will closely follow their definitions, but will use the term context and concept interchangeably:

Definition 3 (Taxonomy). Concept descriptions in L are formed using the following syntax rule:

$$C, D \rightarrow AC \mid \top \mid \bot \mid \neg C \mid C \sqcap D \mid C \sqcup D$$

where AC denotes an atomic concept.

Atomic concepts are those concepts that are directly associated with sensor readings, and are not defined in terms of other concepts. For instance, we can define the atomic concepts *Smokey*, *Clear*, *Smelly* and *Fresh*. The first two concepts can be directly observed by a smoke detector and the last two can be observed by chemical sensor. We could then construct derived concepts, such as *Murky_Air* = *Smokey* ⊓ *Smelly*.

Apart from describing concepts we need to have a way to interpret these concepts, as this will allow us to interpret the state of the space u ∈ U. Hence we define concepts in terms of U:

Definition 4 (Interpretation Function). *We define the interpretation I as follows:*

$AC^I \subseteq U$ $\quad \top^I = U$ $\quad \bot^I = \varnothing$

$(\neg C)^I = U - C^I$ $\quad (C \sqcap D)^I = C^I \cap D^I$ $\quad (C \sqcup D)^I = C^I \cup D^I$

We would also like to specify how a context or group of contexts relates to another. There are two ways of declaring relationships between two contexts. One is the inclusion relation, where some base context is a component of a larger composite context (e.g., the base contexts *Smoky Air* and *Foggy* are included in the composite context *Smoggy*). The other is the equivalent relation, used to resolve ontological issues where the same context has multiple names. Formally these relations are defined as:

Definition 5 (Context Relations). Inclusion and equality are defined as follows:

Inclusion: $C \sqsubseteq D$, *interpreted by* $C^I \subseteq D^I$

Equality: $C \equiv D$, *interpreted by* $C^I = D^I$.

The description of how a set of concepts relate to each other is called a terminology or T–Box. A terminology *T* is a collection of inclusions and equalities that define how a set of concepts relate to each other. Each item in the collection is unique and acyclic. Concepts that are defined in terms of other concepts are called derived concepts. An interpretation *I* that interprets all the atomic concepts will allow us to interpret the entire taxonomy.

Interpretation of the current state of the world is straightforward. We identify whether the current state of the world is a member of any concept. In other words, to verify whether $u \in U$ satisfies concept *C* we check that $u^I \in C^I$. The interpretation of the current state of the space leads us to the current active context:

Definition 6 (Active Context). The active context of the space R: U → 2^C is:

$$R = \{C \mid u^I \in C^I i\}$$

Much of the related research takes different approaches to derive higher level information. Most sensor networks in the literature make use of a hierarchy of interpretation functions, often referred to as context derivation [16, 17, 18], which take as input the readings (or history of readings) from sensors and produce a new output value. We avoid these derivations because there is no guarantee that the derived functions will follow the consistency rules as specified by description logic. The inconsistency implies that contradictory or invalid (undefined) context states could arise, introducing potentially dangerous situations.

5.2 Controlling Actuators

Apart from observing the space, we are also controlling the space using actuators. Within a smart space, users and the system can perform sequences of actions by turning actuators on or off.

Definition 7 (Statement Sequence). Given a set of actuators *A*, a statement sequence *S*, also known as a sequence of actions, is defined as follows:

$$S ::= \uparrow a_i \mid \downarrow a_i \mid S1; S2$$

Where $\uparrow a_i$ turns actuator $a_i \in A$ on, and $\downarrow a_i$ turns actuator a_i off. We also denote action prefixing using the symbol ';' to indicate that we first perform statement S1 and after which we execute the remaining statement S2.

Using the description of an intentional effect it is now straightforward to identify whether or not two actuators conflict with each other.

Definition 8 (Conflicting Actuators). Two actuators a_i and a_j are in conflict in context *u* if $g_i(u) \cap g_j(u) = \varnothing$

That is, the invocation of a_i leads to a state of the world that is disjoint from the state to which the invocation of a_j will lead, hence this state of the world can never arise. A classical example of the invocation of two conflicting actuators is the air-conditioning and the heater. A heater will raise the room temperature whereas an air-conditioner will lower the temperature, hence a statement that activates both the air-conditioner and the heater at the same time is considered to be contradictory.

5.3 Beliefs-Desires-Intentions of Users

As we mentioned earlier a user has a belief about the possible states of the world *T*, which can be captured by taxonomy and the interpretation of the user. More The current belief about the state of the world can be captured with taxonomy and an accompanied interpretation. The interpretation maps the values obtained from a sensor to the atomic concepts. We express the intentions *I* of the users by a sequence

of actions the user wishes to execute in a particular context. Formally this is modeled as:

Definition 9 (User). A user can be denoted by a tuple

$$B ::= <T ; I; D, X, I>$$

Where T and I are the taxonomy accompanied by the interpretation of the user. The user can then specify desired contexts D, and extremely undesirable impermissible contexts X. Let I: C → S be a mapping from context to statements that shows possible intentions to transit away from context C.

Definition 10 (Active and Passive Intentions): Assume current active context is represented by R the following transitions defined the activeness and passiveness of intentions:

Active Intention: $I_a : R \xrightarrow{I_a} D$

Passive Intention: $I_p : D \xrightarrow{I_p} R$

For instance, turning on a ventilation fan is an action that has an intentional effect of transiting the air quality in a house from *Murky Air* to *Clean Air*, therefore "Turn On Vent" is an active intention. Turning on the stove to sear steak, on the other hand, can produce smoke and degrade air quality from *Clean Air* to *Murky Air*. By turning off the stove we may eliminate the source of the smoke, therefore there is a possibility for improving the air quality over time, hence "Turn Off Stove" is a passive intention. In our model, the active intentions (what happens when an actuator is turned on) are specified by programmers, and the passive intentions (what happens when an actuator is turned off) are inferred by the system.

5.4 The Smart Space Programming Model

We can now define a space consisting of sensors and actuators and a user using the following operational semantics:

Definition 11 (Programmable Pervasive Space). A programmable pervasive space can be denoted by a tuple consisting of:

$$P ::= <B;R; S; 2^A>$$

Where B is the representation of the user, R the currently active context, S the statements we are currently processing and 2^A *the set of actuators currently active.*

Spaces changes over time due to nature. We model the effect of nature by a transition that changes the state of the space, and observe the changes of the space through our sensors.

Definition 12 (Environmental Effect).

$$<b; f(u); S; a> \rightarrow <b; f(u'); S; a> \text{ where } f(u) \neq f(u')$$

The other way in which the space can change is by actuators that the space controls. The following transitions capture the turning on and off of actuators:

Activation: $<b; u; \uparrow ai; a> \rightarrow <b; u; \varepsilon ; a \cup ai>$

Deactivation: $<b; u; \downarrow ai; a> \rightarrow <b; u; \varepsilon; a \setminus ai>$

If the set of active contexts has changed, we obtain the intentions from the user and execute the desired sequence of actions:

Context Change: $<b; f(u); S; a> \rightarrow <b; f(u'); i(R(f(u'))); a>$ whenever $R(f(u')) \neq R(f(u))$

6 Scenario and Programming Procedures

The context-driven model defined in the previous sections, when comparing to other existing programming models for pervasive computing, provide stronger safety features and better capability to evaluate multiple potential action plans. This model, as previously discussed, is roughly based on the belief-desire-intention model. The overall state of a smart space is captured in a context graph based on the interpretation of the user (*T*, *I*), in which the desirable contexts (*D*) and the impermissible contexts (*X*) are specified. The sensors retrieve the readings, and identify the current active contexts (*R*), and then the smart space devises the plan *I* to move the state from *R* toward *D*. We next demonstrate three advantages of this context-driven model using the following scenario.

6.1 Applying the Context-Driven Programming Model

On a chilly night in December, Matilda is searing a rib eye steak on the stovetop for dinner. Her steak is well-marbled with fat, and is generating a lot of smoke as it cooks. The smoke clouds up the kitchen, and soon sensors detect that the air quality in the house is fast degrading.

According to the context diagram of the house (Fig. 1), there are two contexts in the "Air Quality" domain ($D_{air\_quality}$). They are respectively known as the *Murky Air* context and the *Clean Air* context, with *Clean Air* being the preferred context. The air quality can be monitored using a Smoke Detector Sensor, represented in our model as the function:

$f_{smoke\_detector} : U \rightarrow D_{air\_quality}$ where *U* represents the overall state of the house.

The current state of the house can be represented by the active context of the space R, which consists of the contexts that can be observed and are currently true. Before Matilda starts to sear the steak, *R* = {*Clean Air*, *Low Power Draw*, *Warm Indoors*}. At some point, because of Matilda's cooking, *Murky Air* has replaced *Clean Air* in *R*.

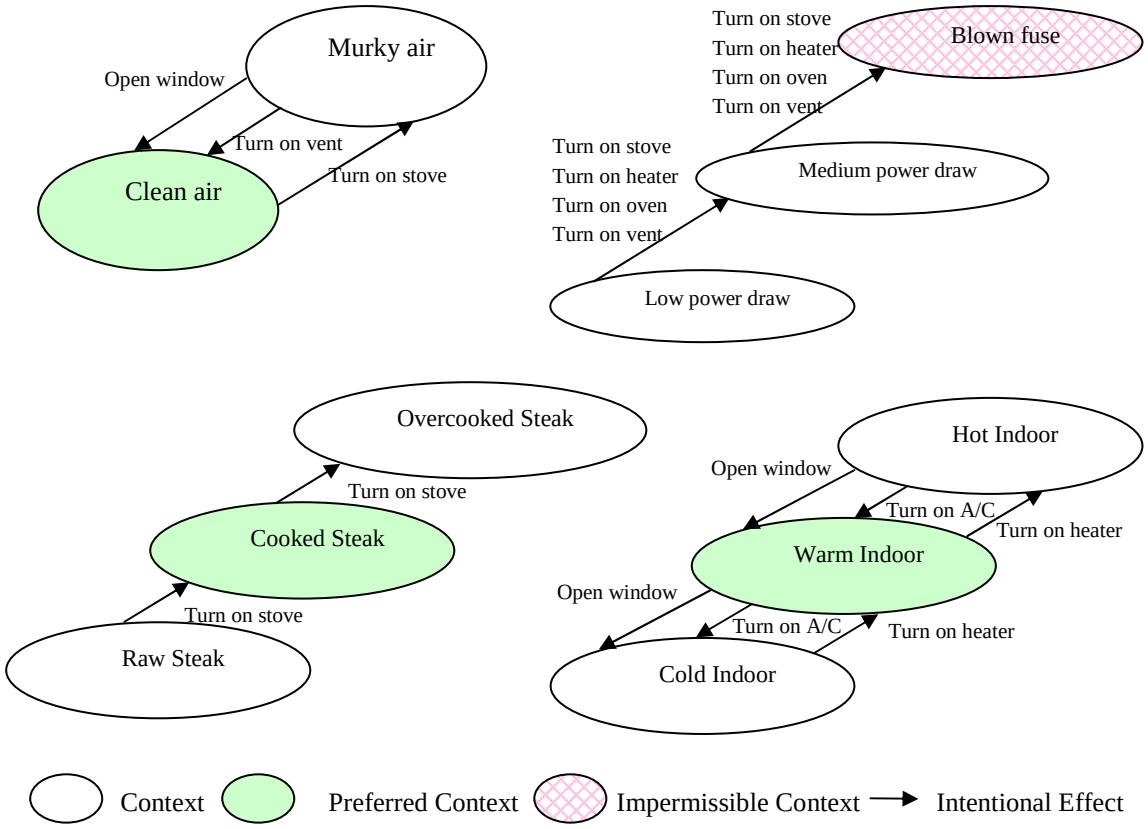

Fig. 1. Context Graph for Matilda's House

Looking up the context graph, as shown in Figure 1, the home server found three possible courses of actions that can be employed to improve the quality of the air:

1. Open the window
2. Turn on the vent
3. Stop the cooking by turning off the stove

For this particular example, there are three actuators of interest. Therefore we define the actuator set A as {*Window*, *Vent*, *Stove*}, because at least one of their intentional effects are relevant to air quality. In particular, "Open Window" and "Turn On Vent" can both take us actively from the *Murky Air* to *Clean Air* context, while "Turn On Stove" has the reverse effect, therefore it is possible that by turning *off* the stove, we can observe the improvement of air quality. These intentional effects are described in our model as:

$g_{window}^{air\_quality}$: *Murky air* → *Clean air*

$g_{vent}^{air\_quality}$: *Murky air* → *Clean air*

$g_{stove}^{air\_quality}$: *Clean air* → *Murky air*

In this case, "Open Window" and "Turn on Vent" are active potential actions because they can proactively improve the air quality, while "Turn Off Stove" is a passive option.

6.1.1 Evaluating Potential Actions

When evaluating all the possible options, the home server first examines the active options before considering passive options. It starts by checking the side effects of each of the potential actions. In this case, it finds that, in addition to improving the air quality, opening the windows may cause the room temperature to drop if the outdoors temperature is lower than the current room temperature. It also finds that turning on the vent will increase the power load of the house. These side effects are represented as:

$g_{window}^{temperature}$: *Hot Indoors* → *Warm Indoors; Warm Indoors* → *Cold Indoors*

$g_{vent}^{power\_draw}$: *Low Draw* → *Medium Draw; Medium Draw* → *Blown Fuse*

Upon calculation, the home server decides that opening the window would cause the room temperature to drop from warm to cold, which is much less preferable, while turning on the vent will only increase the power draw marginally. Hence the home server decides that turning on the vent is the preferable action. In other words, the home server will now choose an action plan consisting of the statement $S = \{\uparrow vent\}$ in an attempt to improve air quality in the house to *Clean Air*.

6.1.2 Detecting Conflicting Directives

A few minutes pass, but the sensors have not reported any significant improvement in air quality. The vent can barely keep up with the smoke produced by searing the steak. The context *Murky Air* is still part of the active context R, and the home server must employ another means to clear the air. Its only remaining option is to turn off the stove ($S_1=\{\downarrow stove\}$) so as to stop the steak from producing more smoke. However, this is in conflict with the cooking assistance service, which wants to keep the stove on high heat ($S_2=\{\uparrow stove\}$) until the internal temperature of the steak reaches 133 ˚F.

As we have defined, a programmable pervasive space is represented as the 4-tuple $\{B; R; S; 2^A\}$. At this moment, S includes two separate statements: S_1 (trying to deactivate the stove actuator) and S_2 (trying to activate the same stove actuator). Detecting that S_1 and S_2 are contradictory directives, the home server prompts Matilda to see if she prefers to turn off the stove to improve the air quality or to leave the stove on until the steak is done.

6.1.3 Avoidance and Handling of Impermissible Contexts

Matilda decides that just steak isn't enough for dinner, and prepares to toast some garlic bread in the oven. She would of course like to toast the bread while the steak is searing, so everything is done at the same time.

On this chilly night, however, the heater in Matilda's house is already running at full force. With the stove already turned on to high heat and the vent whirling to keep the smoke out of the kitchen, the home server calculates that turning on the oven would draw too much power, resulting in an impermissible context, *Blown Fuse*. In such a cold night, Matilda's frail condition, a blown fuse that cut off electricity would greatly endanger her health. The home server therefore decides to prevent oven from being turned on, and announce a message to Matilda through the speaker in the kitchen to try again later.

6.2 Programming Procedures

How do programmers actually program a pervasive computing space using context-driven programming model? We identified the following three-step procedure:

1. **Design the Context Graph**: Programmers have to decide what the domains of interest are, and what the contexts of interest within these domains are. This decision is heavily influenced by availability of the sensors, the services planned, and the users' belief and desires.
2. **Interpret Sensor Readings**: Programmers have to define interpretation functions from ranges or enumerated possible reading values from various sensors to atomic contexts appearing in the context graph.
3. **Describe Intended Behaviors**: Programmers have to describe intended behaviors in terms of action statements associated with each context in the context graph, so that smart space knows which actuators to manipulate when various contexts become active

A prototype IDE has been implemented to facilitate the programming practice using this context-driven model. The prototype is currently being tested internally.

7 Conclusion

Pervasive computing systems must interact with and influence the physical world, querying and controlling a large number of transient devices, as well as meeting the range of needs of users. Special mechanisms are therefore needed to handle such diversity. Many research projects have explored different programming models for pervasive computing, but the applications developed can only focus on achieving their own goals and often lose sight of the overall picture in the smart space. This can lead to conflicts and erroneous behaviors. Building upon the experience learned during the implementation of the Gator Tech Smart House, we proposed and experimented with a new context-driven programming model to address some of these shortcomings.

In other existing models, contexts only complement the traditional programming; our model uses contexts as the primary building blocks. Programmers build the

context graph that captures all possible states that are of interest in the smart space. Contexts from the graph are marked as desirable, transitional, or impermissible. Programmers also define the intentional effects of actuators in terms of transitions from one context to another. The system is responsible for identifying active contexts from sensor readings and for choosing actions that can lead to more desirable contexts.

The benefits of our context-driven programming model are improved coordination using explicitly defined behaviors based on context, enhanced safety guarantees and real-time, at-a-glance snapshots of the space's status. By explicitly describing the effect of all the basic entities we can detect conflicting devices at compile time, and the system is better able to evaluate multiple potential action plans. In addition, this explicit description makes it easier to change services based on users' varying preferences. The model is also able to detect conflicting or dangerous behaviors at runtime. Finally, the explicit context graph allows programmers to define impermissible contexts – states of the space that are extremely dangerous and must be avoided – as well as providing the support at runtime to avoid and handle them.

The formalization of the model and the provided scenario demonstrate the practicality of this model. We are currently developing and testing tooling support for this model, as well as integrating it with other technologies developed in our lab, mainly the Atlas sensor network platform [19], providing an end-to-end solution for creating programmable pervasive spaces.

References

1. A. Dey, D. Salber, G. Abowd, A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction (HCI) Journal 16 (2001) pp 97 – 166
2. T. Gu, H. Pung, D. Zhang, A service-oriented middleware for building context-aware services. Journal of Network and Computer Applications (JNCA) 28 (2005) pp 1 - 18
3. H. Chen, T. Finin, A. Joshi, F. Perich, D. Chakraborty, L. Kagal, Intelligent agents meet the semantic web in smart spaces. IEEE Internet Computing 8 (2004)
4. T. Gu, H. Pung, D. Zhang, Toward an OSGi-Based Infrastructure for Context-Aware Applications, In IEEE Pervasive Computing, Oct - Dec 2004, pp 66 -74
5. R. Bose, J. King, H. El-zabadani, S. Pickles, A. Helal, Building Plug-and-Play Smart Homes Using the Atlas Platform, Proceedings of the 4th International Conference on Smart Homes and Health Telematic (ICOST), Belfast, the Northern Islands, June 2006.
6. A. Helal, W. Mann, H. Elzabadani, J. King, Y. Kaddourah and E. Jansen, Gator Tech Smart House: A Programmable Pervasive Space, IEEE Computer magazine, March 2005, pp 64-74.
7. G. Chen, D. Kotz, A survey of context-aware mobile computing research. Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College (2000)
8. G. Abowd, A. Dey, P. Brown, N. Davies, M. Smith, P. Steggles, Towards a better understanding of context and context-awareness. Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing, Springer-Verlag (1999) pp 304 - 307
9. J. Hong, J. Landay, An infrastructure approach to context-aware computing. Human-Computer Interaction 16 (2001)
10. D. Salber, A. Dey, G. Abowd, The context toolkit: Aiding the development of context-enabled applications. CHI. (1999) pp 434 - 441

11. M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. Campbell, K. Nahrstedt, Gaia: A Middleware Infrastructure to Enable Active Spaces. In IEEE Pervasive Computing, Oct-Dec 2002, pp 74-83
12. C. Greenhalgh, S. Izadi, J. Mathrick, J. Humble, I. Taylor, ECT: A Toolkit to Support Rapid Construction of UbiComp Environments, Online Proceedings of the System Support for Ubiquitous Computing Workshop at the Sixth Annual Conference on Ubiquitous Computing (UbiComp 2004), September 2004
13. The OWL Services Coalition, OWL-S: Semantic markup for web services. http://www.daml.org/services/owl-s/1.0/owl-s.html (2004)
14. M. Wooldridge, Reasoning about Rational Agents. The MIT Press, Cambridge, Massachussetts/London, England (2000)
15. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. Patel-Schneider, eds., The Description Logic Handbook. Cambridge University Press (2002)
16. G. Chen, D. Kotz, Solar: An open platform for context-aware mobile applications. an informal companion volume of short papers of the Proceedings of the First International Conference on Pervasive Computing. (2002) 41 - 47
17. N. Cohen, H. Lei, P. Castro, A. Purakayastha, Composing pervasive data using iql. Proceedings of the Fourth IEEEWorkshop on Mobile Computing Systems and Applications, IEEE Computer Society (2002) 94
18. R. Kumar, M. Wolenetz, B. Agarwalla, J. Shin, P. Hutto, A. Paul, U. Ramachandran, Dfuse: a framework for distributed data fusion. In: Proceedings of the first international conference on Embedded networked sensor systems, ACM Press (2003) 114 – 125
19. J. King, R. Bose, H. Yang, S. Pickles, A. Helal, Atlas: A Service-Oriented Sensor Platform, To appear in the Proceedings of the First International Workshop on Practical Issues in Building Sensor Network Applications (in conjunction with LCN 2006), November 2006