Physical Interaction in Pervasive Computing: Formal Modeling, Analysis and Verification

Fuyuki Ishikawa
GRACE Center, National
Institute of informatics, Tokyo, Japan
f-ishikawa@nii.ac.jp

Basem Suleiman
National ICT Australia
(NICTA), Sydney, Australia
University of New South Wales, Sydney, Australia
basem.suleiman@nicta.com.au

Kayoko Yamamoto
The University of Tokyo, Tokyo, Japan
kymmt@nii.ac.jp

Shinichi Honiden
GRACE Center, National
Institute of informatics, Tokyo, Japan
honiden@nii.ac.jp

ABSTRACT
Application software in pervasive computing is required to control devices embedded in the environment by being aware of the contexts on which effectiveness of the devices depend. Developers face difficulties to enumerate involved physical prerequisites for effective use of devices and undesirable situations to be avoided, as well as to define consistent behaviors of the application software. This study provides a theoretical framework for formal modeling of requirements, assumptions and behaviors for application software in pervasive computing. This study specifically focuses on prerequisites for physical (visual, audio, etc.) interactions, which are defined and examined in terms of scopes and their relationships not limited to tree structures. This study also explores analysis and verification based on the formal modeling, using of an existing reasoner.

Categories and Subject Descriptors
D.2.1 [Software Engineering]: Requirements/Specifications

General Terms
Languages

Keywords
Pervasive Computing, Formal Specifications, Event Calculus

1. INTRODUCTION
The notion of pervasive computing has attracted widespread attention, primarily targeting user support by often invisible (unnoticed) use of sensing or actuating devices embedded in the surrounding environment [16]. A variety of applications has been investigated, from smart homes to health care, typically making use of physical (e.g., visual, audio, tactile, or radio) interactions between users and embedded devices.

These physical interactions require specific conditions to hold before the interactions start and while they last, in order to have their meaningful effects. Such prerequisites have been modeled in terms of scopes, e.g., “inside the room to hear sounds from the speaker” and “near by the display to use its touchscreen” [15]. Such prerequisites for physical interactions should be taken into consideration, when behavior of the application software is discussed. For example, application software involving audio interaction may include behaviors such as “hand and activate a speaker in the room in which the user currently is” and “stop streaming when the user is getting out of the room.” For effective use of devices and user support, developers need to enumerate, define and specify requirements and application behaviors related to involved physical interactions. Such specifications need to be collectively exhaustive, as well as precise and unambiguous. However, this task is costly and difficult for developers, especially when tackled from scratch.

In addition, it is very significant to clarify required properties and ensure they hold, e.g., confidential documents are never being shown on a display when an unauthorized person is able to see it. Satisfaction of such a property depends on application behaviors in each situation, e.g., denying user request to display if there is an unauthorized person in the same room. It also depends on environmental assumptions, e.g., whether there is a way to physically prevent unauthorized persons from entering the room. In this way, developers need to carefully clarify required properties as well as enumerate and determine related behaviors and assumptions. Developers also need to adjust and refine them through analysis and verification so that they are consistent. As the environments are typically very dynamic, it is often too difficult, without help of tools, to verify and ensure that required properties hold even with existence of a variety of situations.

In response to the above issues, this study proposes a theoretical framework for formalization and reasoning of phys-
Figure 1: A Meeting Support System

As a typical application in pervasive computing, it involves use of devices, carefully controlled by the application software (often without detailed user instructions) depending on contexts. The example system involves the following kinds of physical interactions between users and devices.

Visual Users see slides on the main screen, see demonstration on each display along the wall, etc.

Audio Users hear presentation, discussion by the attendees, etc.

Tactile Users touch screens of demonstration displays to operate them, touch keyboards to operate the slides, etc.

Each of these physical interactions has a prerequisite for its meaningful effect, i.e., a user is enough near by to interact with (see, hear, touch, etc.) a device. Application software is often required to be smart by being aware of this kind of prerequisites.

Below describes examples of detailed behavior specifications related to a prerequisite for visual interaction with a display.

B1 Given a request by an authorized user in the room (who is allowed to access the content and can see the display), the application starts to show information required for the conference on the display in a room.

B2 It stops showing the confidential information when an unauthorized user comes into the room (and becomes able to see the display).

B3 It shuts down the display when all the users get out (and become unable to see the display).

These behaviors are derived from high-level requirements of the following, respectively.

R1 Information of the conference is shown on the display in the room when an authorized user in the room requests.

R2 Confidential information is never kept able to be viewed by an unauthorized user.

R3 The display is never kept active when no one can physically interact with it.

Introduction of the behavior B2 assumes there are possibilities unauthorized users enter the room. In the case it is possible to physically prevent unauthorized users from entering the room, the assumption should be clearly specified and used for analysis.

Below discusses two difficulties in development of application software in pervasive computing through these examples.

First, it should be facilitated for developers to enumerate, define and specify such requirements and behaviors in early stages in development. Especially, it is necessary but difficult to make them collectively exhaustive as well as precise and unambiguous. Besides general techniques in software engineering for this issue (e.g., systematic analysis methods), further support is desirable specific to and common in pervasive computing.

Specifically, it is useful to provide modeling of physical interactions as well as related requirements and behaviors.
including vocabularies as well as structural and semantic constraints and patterns. Developers can make use of the modeling and focus on their own requirements and behaviors, avoiding starting from scratch. The above example involves essential aspects in application software for pervasive computing, as follows.

- The requirement R3 seems common for any kind of physical interactions and devices, and can be generalized and commonly used.
- All the requirements and behaviors mention the prerequisite for visual interaction with the display, which can be generalized and commonly used.
- The requirement R2 and the corresponding behavior B2 mention reaction to incursion of unauthorized users, or resolving undesirable situations (it is said “undesirable situations are never kept”). This should be clearly defined and distinguished from prior prevention. Such handling patterns of undesirable situations (prevent, react to and resolve, or disregard) can be commonly used as well.

It is notable that formal modeling with strict syntax and semantics helps (or enforces) developers exclude unclarity and unambiguity. Formal modeling also enables tool-supported analysis and verification, which is discussed below.

Second, it should also be facilitated for developers to analyze and verify requirements and application behaviors in order to ensure they are consistent. In the above example, the requirement R2 is not ensured only by the behavior B2 because B2 does not mention situations where there are already unauthorized users in the room. Actually, the behavior B1 breaks R2 by starting to show the information on the display even if there are unauthorized users. As R2 is a security requirement with high priority, R1 and B1 would be adjusted.

R1’ Information of the conference is shown on the display in the room when an authorized user requests and there is no unauthorized user in the room.

B1’ Given a request by an authorized user, the application starts to show information required for the conference on the display in a room if there is no unauthorized user in the room. Otherwise, the application denies the request regardless of the permission given to the requester.

Specifically, it is very difficult to explore possible situations in a collectively exhaustive way so that consistency is ensured with existence of a variety of situations under uncontrollable, dynamic environments. Formal modeling and its effective use should be explored to allow for tool-supported analysis and verification. Especially model checking is effective means for the above problem, which exhaustively examines possible situations and checks satisfaction of properties. Other analysis based on formal modeling is also useful, such as simulation of a particular event occurrence and discovery of event occurrences leading to a particular situation, as well as visualization of such analysis.

2.2 Modeling Approach

As discussed in Section 2.1, the objective of this study is to provide formal modeling of requirements, assumptions and behaviors for application software in pervasive computing. This study specifically focuses on prerequisites for physical interactions, and also expects the formal modeling to be the basis of analysis and verification.

2.2.1 Scopes

Regarding prerequisites for physical interactions, [15] used modeling of scopes, called “Aura Components”, in which a user can interact with (see, hear, touch, etc.) each device effectively. The work discussed monitoring and reacting to runtime events, such as user movement, not for analysis and verification tasks at the development phase. The work used a tree-based structure to denote relationships between the scopes, as many studies on location models did [14].

This study also models prerequisites for physical interactions in terms of scopes. Such scopes are defined for each kind of physical interaction with each device, in order to, for example, allow for specifying “can see but cannot touch the device.”

This study expects developers to model scopes meaningful for the application, in an abstract way, in early stages of their development processes. For example, the examples in Section 2.1 mentioned scopes, for visual interaction with a display, roughly as “in the room or not” (Figure 2 (a)). It implies that the application is required to be aware of the necessary condition, “users need to be in the room to see the display.” It implies that the application is not required to be aware of the more detailed condition, “users need to be at specific positions to see the display” (because it would be bothersome if the application warns about that point). On the other hand, for demonstrations, each with its own device, it is necessary for the application to distinguish which device the user is standing in front of. Then the scopes are defined in a more granular way (Figure 2 (b)).

2.2.2 Scope Overlaps

In addition, this study introduces modeling of scope overlaps. For example, regarding audio interactions, application software should not activate multiple speakers whose sounds can reach a user at the same time and make them ineffective (just noisy). To handle (model) such constraints and related behaviors, this study explores modeling of whether scopes are disjoint or they overlap, which does not naturally appear in tree-based modeling.

In the example in Figure 2, scopes would be defined similarly for presentation and demonstration. This time application software should not activate audio functionalities of deviceA and any of device1 or device2 simultaneously, while it may activate device1 and device2 simultaneously.

If the developers would easily introduce application behaviors such as the following one, it will lead to undesirable
situations where users find sounds from multiple speakers noisy.

(anytime) activate any speaker when a user becomes able to hear sounds from it

Explicit modeling of scope overlaps enables reasoning on this kind of inconsistency.

To allow for such modeling, this study considers non-tree-based modeling of scopes and their relationships. Another discussed example, where modeling of scope overlap is thought to be useful, is “allocate some of the public displays to users so that each user cannot see any other’s display (for privacy information, examination answers, or customized advertisements).”

3. FORMAL MODELS FOR PHYSICAL INTERACTION

This section describes the proposed modeling and its representation in Event Calculus.

3.1 Event Calculus

Event Calculus is a well-known formalism for expressing and reasoning about effects of actions [17]. Event Calculus considers predicates called fluents whose boolean value can change as effects of actions or time passage, and provides definitions and axioms to specify and reason about how fluent values change. For example, Initiate($act$, $flu$, $t$) denotes the fact that occurrence of action $act$ at time $t$ makes fluent $flu$ to start to hold. So the following expression defines an axiom that says when a user enters a room then a user becomes to be in the room (for any user, room, and anytime).

$$\forall user, room, t 
\text{Initiate(Enter(user, room),}$$
$$\text{In(user, room), t)}$$

On the other hand, the predicate Terminate is used to denote the counterpart, when a user exits a room then a user becomes not to be in the room.

$$\forall user, room, t 
\text{Terminate(Enter(user, room),}$$
$$\text{In(user, room), t)}$$

Fundamental predicates defined in Event Calculus are shown in Table 1.

In Event Calculus axioms, intuitively, fluent values can change only when they are explicitly initiated or terminated by actions (through the Initiate and Terminates predicates) or they have been declared to have random values (through the Release predicate). On the basis of such axioms, Event Calculus allows for reasoning about all the possibilities of how actions happen and fluent values change through time passage, which satisfy all the given statements.

3.2 Models

This study provides vocabularies, or definitions of events, fluents and their axioms as well as meta expressions (aliases for Event Calculus expressions). Below describes different types of vocabularies defined and provided in this study.

3.2.1 Scopes

Scopes are specified by a fluent InScope, which are affected by events EnterScope and ExitScope. Below defines two axioms between the fluent and each of the events, which are used commonly for any environments or applications.

\begin{align*}
\forall user, device, itype, t 
\text{Initiate(EnterScope(user, device, itype),} \\
\text{InScope(user, device, itype), t)} \\
\forall user, device, itype, t 
\text{Terminate(EnterScope(user, device, itype),} \\
\text{InScope(user, device, itype), t)}
\end{align*}

Here itype means the type of physical interactions, such as visual, audio, and tactile.

The following meta expression DisjointScopes can be used to specify a fact that two scopes never overlap.

\begin{align*}
\text{DisjointScopes(device1, device2, itype) } \equiv \\
\forall user, t 
\text{HoldsAt(InScope(user, device1, itype), t)} \\
\rightarrow \neg \text{HoldsAt(InScope(user, device2, itype), t)}
\end{align*}

It is also possible to define containment relationships (like in trees-structures).

\begin{align*}
\text{ScopesIncluded(device1, itype1, device2, itype2) } \equiv \\
\forall user, t 
\text{HoldsAt(InScope(user, device1, itype1), t)} \\
\rightarrow \neg \text{HoldsAt(InScope(user, device2, itype2), t)}
\end{align*}

These expression can be used by developers to define settings specific to environments or applications.

3.2.2 Other Fundamental Vocabularies

Besides the scopes, several fundamental vocabularies are introduced. Part of them are shown in Table 2. Locations are modeled similarly to the scopes. Activation and deactivation of devices are modeled, distinguished from requests for activation and deactivation. Related axioms and meta expressions are omitted as they can be defined similarly to those of scopes.
actions

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnterLocation (user, loc)</td>
<td>user enters location loc</td>
</tr>
<tr>
<td>ExitLocation (user, loc)</td>
<td>user exits location loc</td>
</tr>
<tr>
<td>Activate (dev, itype, con)</td>
<td>activate a functionality for physical interaction of type itype (e.g., visual) on dev (e.g., display1) with con (e.g., content1)</td>
</tr>
<tr>
<td>Deactivate (dev, itype, con)</td>
<td>deactivate a functionality for physical interaction of type itype on dev with con</td>
</tr>
<tr>
<td>ActivateRequest (user, dev, itype, con)</td>
<td>user requests to activate a functionality for physical interaction of type itype on dev with con</td>
</tr>
<tr>
<td>DeactivateRequest (user, dev, itype, con)</td>
<td>user requests to deactivate a functionality for physical interaction of type itype on dev with con</td>
</tr>
</tbody>
</table>

fluent

<table>
<thead>
<tr>
<th>Fluent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InLocation (user, loc)</td>
<td>user is in location loc</td>
</tr>
<tr>
<td>Active (dev, itype, con)</td>
<td>a functionality is active for physical interaction of type itype (e.g., visual) on dev (e.g., display1) with con (e.g., content1)</td>
</tr>
</tbody>
</table>

Table 2: Other Fundamental Vocabularies

3.2.3 Application Behaviors

Application behaviors may include sequential/concurrent processes and event handlers. In both cases, an action becomes ready by action occurrences (completion of the previous action(s) or event occurrences). An action may not be executed immediately when it becomes ready. An expression pattern of the following is provided to model this nature.

\[
\forall t, \text{ all the free variables in } act_1, act_2, \text{ cond} \\
\text{Initiates}(act_1, act_2, t) \\
\Leftarrow \text{HoldsAt}(\text{cond}, t) \\
\forall t, \text{ all the free variables in } act_1, act_2, \text{ cond} \\
\text{Terminates}(act_2, act_2, t) \\
\forall t, \text{ all the free variables in } act_1, act_2, \text{ cond} \\
\text{HoldsAt}(\text{act}_2, \text{Ready}, t) \Leftarrow \text{Happens}(\text{act}_2, t)
\]

Here \text{act}_2\text{Ready} is a fluent that has the same arguments as \text{act}_2. This set of statements defines an ECA (Event-Condition-Action) rule, where \text{act}_1 under \text{cond} activates \text{act}_2. It says \text{act}_1 makes \text{act}_2 ready when \text{cond} holds, \text{act}_2 makes \text{act}_2\text{unready}, and \text{act}_2 can happen only when it is ready. Although this definition is naive in a sense it ignores multiple occurrences of \text{act}_1 when \text{act}_2 is ready, one-to-one correspondence between \text{act}_1 and \text{act}_2 can be defined with some more complexity.

Below is the definition of the example behavior based on the above pattern: stops showing the confidential information when an unauthorized user comes into the scope of the display.

\[
\forall \text{display}, \text{user}, \text{con}, t \\
\text{Initiates}(\text{EnterScope}(\text{user}, \text{display}, \text{visual}), t) \\
\Leftarrow \text{HoldsAt}(\text{Active}(\text{display}, \text{visual}, \text{con}), t) \\
\land \\
\neg \text{HoldsAt}(\text{Authorized}(\text{user}, \text{con}), t)
\]

\[
\forall \text{display}, \text{user}, \text{con}, t \\
\text{Terminates}(\text{Deactivate}(\text{display}, \text{visual}, \text{con}), t) \\
\Leftarrow \text{Happens}(\text{Deactivate}(\text{display}, \text{visual}, \text{con}), t)
\]

\[
\forall \text{display}, \text{user}, \text{con}, t \\
\text{HoldsAt}(\text{Deactivate}(\text{display}, \text{visual}, \text{con}), t)
\]

It is notable that it is possible to define nondeterministic behaviors by making two actions ready but not allowing both of them to happen.

3.2.4 Properties

Some definitions are provided to denote typically undesirable situations. For example, below defines a fluent \text{IneffectiveActivation} that becomes true when a device is activated to provide some physical effect but the target user is not in its scope.

\[
\forall \text{user}, \text{device}, \text{itype}, \text{content}, t \\
\text{HoldsAt}(\text{IneffectiveActivation}(\text{user}, \text{device}, \text{itype}), t) \\
\Leftarrow \\
\neg \text{HoldsAt}(\text{InScope}(\text{user}, \text{device}, \text{itype}), t) \\
\land \\
\text{HoldsAt}(\text{Active}(\text{device}, \text{itype}, \text{content}), t)
\]

Another situation is activation of two audio devices whose sounds reach a user, conflict with each other when they are played simultaneously.

\[
\forall \text{user}, \text{device}_1, \text{device}_2, \text{content}_1, \text{content}_2, t \\
\text{HoldsAt}(\text{AudioConflicts}(\text{user}, \text{device}_1, \text{device}_2), t) \\
\Leftarrow \\
\text{HoldsAt}(\text{InScope}(\text{user}, \text{device}_1, \text{audio}), t) \\
\land \\
\text{HoldsAt}(\text{InScope}(\text{user}, \text{device}_2, \text{audio}), t) \\
\land \\
\text{HoldsAt}(\text{Active}(\text{device}_1, \text{audio}, \text{content}_1), t) \\
\land \\
\text{HoldsAt}(\text{Active}(\text{device}_2, \text{audio}, \text{content}_2), t)
\]

The following property, defined as a meta expression, mentions that such a situation needs to be prevented (before it happens).

\[
\text{NeverHolds}(\text{flu}) \equiv \\
\forall t, \text{ all the free variables in } \text{flu} \\
\neg \text{HoldsAt}(\text{flu}, t)
\]
3.2.5 Assumptions

Assumptions can be specified in the same way as properties. The difference between a (required) property and an assumption resides in developers’ viewpoint. The former is target of verification, expected to hold only by proper behaviors, while the latter is considered in advance to hold. An example assumption mentioned in Section 2.1 describes no occurrence of an event: “no unauthorized person can enter the room.” This kind of property can be defined similarly.

\[
\text{NeverHappens}(act) \equiv \\
\forall t, \text{ all the free variables in } act \\
\neg \text{Happens}(act, t)
\]

4. DEVELOPMENT SUPPORT

4.1 Development Tasks

This study aims at supporting development tasks, involving the following.

- Clarify environmental assumptions, especially, involved devices and physical interactions with them.
- Identify, enumerate and specify requirements and assumptions, as well as application behaviors and properties expected to hold.
- Run simulation and analysis and understand the state transitions in the system. Especially, check consistency, including whether properties hold or not, by exploring a variety of situations caused by uncontrollable events (e.g., user movement).
- Add assumptions and constraints when too many possibilities are obtained and they are expected not to appear actually.
- Adjust specifications when inconsistency is found or properties are turned out not to always hold. Typically, strengthen assumptions and constraints in application behaviors in order to meet requirements, or weaken requirements in order to avoid unachievable requirements.

Here it is essential to distinguish what are controllable and what are not. Among the actions defined in Section 3.2, some, such as Activate, are controllable, while others, such as EnterLocation and ActivateRequest, are not controllable. Developers need to carefully make decisions regarding controllable actions (i.e., application behaviors), while varying uncontrollable actions (i.e., external events, especially user actions).

4.2 Reasoning with an Existing Tool

Although it is possible to have manual analysis and verification (theorem proving) on the models in Event Calculus, this study discusses use of an existing reasoner. Discrete Event Calculus Reasoner is one of available reasoners on Event Calculus [2]. It uses the discrete version of Event Calculus where time takes only discrete values (integer numbers). Below is an example of the interpretation in Discrete Event Calculus, where effects of Initiates appear in the next time point.

\[
(\forall t \text{Initiates}(act, flu, t)) \land \text{Happens}(act, 3) \\
\Rightarrow \text{HoldsAt}(flu, 4)
\]

By converting a discrete Event Calculus model into a SAT problem, the tool supports model finding, that is, discovering all models, or possible transitions regarding event occurrences and fluent values through time passage, that satisfy all the given statements. Various reasoning tasks can therefore be supported, for example,

- Input an initial state and a sequence of event occurrences, then simulate what actions happen and what fluents hold. This can be used for simulation or testing purposes in development.
- Input an initial state and a constraint that says an particular property eventually holds, then obtain all the possible models that lead to the situations. This can be used for analysis purposes in development, such as extraction of all the possible paths that lead to undesirable situations.
- Input an initial state, constraint statements to prevent undesirable situations, and a statement that says an undesirable situation is eventually reached, then examine whether there exists any model. This can be used for model checking purposes in development, such as checking whether the constraints are enough to prevent undesirable situations.

5. DISCUSSION

This section discusses advantages and limitations of the presented study as well as related studies.

5.1 Advantages

This paper has discussed formal modeling of requirements and behaviors in application software for pervasive computing. It allows for abstract modeling that especially focuses on prerequisites for physical interactions, which affect activation of effective use of devices or deactivation of their ineffective use. The provided modeling facilitates developers to enumerate and specify requirements, assumptions, and application behaviors by using provided vocabularies including definitions of typical undesirable situations. It also provides foundations for analysis and verification. One of the notable points of the study is modeling of scope overlaps, which allows for examination of conflicts in physical interactions, such as sound conflicts.

On the other hand, the provided modeling abstracts away details of the following.

Deployment The action vocabularies for application behaviors does not include the subject, i.e., it is not necessary to define on which host what kind of components are deployed, and each of which has what kind of responsibilities in the system. So developers can concentrate on observable behaviors of the application software, without deployment issues, which will often be discussed in later phases.

Sensors Physical conditions are described in an abstract way, i.e., in terms of scopes. In actual implementation, outputs from sensors are analyzed and used for context interpretation. With abstract modeling, developers can concentrate on expectations on the application software without caring about detailed configurations and precisions of sensors.
5.2 Limitations

One of the most significant directions is to provide a sophisticated tool. Because this study defines domain-specific, higher-level dedicated vocabularies than Event Calculus itself, it is necessary to provide a proper interface (GUI) that facilitates perception of and operations on the involved concepts (e.g., controllable actions and uncontrollable events).

Regarding the tool, it is also necessary to explore potential improvement in formal analysis and verification. This study provided an initial experience with an existing tool, and left optimization of algorithms or tools as future work.

5.3 Related Work

In the pervasive computing area, models of physical relationships and events, especially formal ones, have been investigated. In [14] and other similar studies, formalisms such as Ambient Calculus [4] was used to model hierarchical relationships of locations and users/devices, as well as other aspects such as execution capabilities. Ambient Calculus is a kind of process algebra suitable for modeling hierarchical structures and their changes. Those studies did not discuss prerequisites for each physical interaction with each device or scope overlaps.

In [15], scopes were modeled in which a user can physically interact with each device effectively. However, that work focused on monitoring and reacting to events by the runtime system, not the development phase and tasks involved there. It did not consider scope overlaps, either.

There have also been many studies that use ontologies to model context information and especially relationships between different vocabularies [13, 5]. These studies thus focus on runtime management of context information, including translation between data from different domains as well as between different layers (e.g., application layer and sensor layer). Ontology-based languages (e.g., RDF) potentially allow for modeling scopes for each type of physical interaction as well as their non-tree-based relationships, though such modeling has not been discussed yet. In addition, ontology-based languages have been used for runtime description of context information at some time point, typically describing the "current" one. On the other hand, this study has focused on analysis and verification tasks at the development phase and explicitly included modeling of event occurrences and successive changes in context information.

This study focuses on early phases of development where implementation details regarding contexts are abstracted away. So this study complements with many studies on data fusion to obtain abstracted context information from sensor data, device states and so on [30, 11].

In general software engineering, formal methods [9] have been used for defining abstract specifications (models) based on strict syntax and semantics, enabling developers to focus on essential aspects of the system while excluding uncertainty and unambiguity. They have also been used for analyzing and verifying the specifications, enabling developers to exclude deficiencies and errors. Especially, model checking methods have been used to exhaustively explore and check all the possible situations (e.g., timing of thread switching) in a given scope [3]. This study has followed these general principles, but investigated an approach specific to pervasive computing.

Formal methods often provide not only sound formalisms but also efficient tools for verification. Although this study provided an initial experience with one of the formalisms and the tools, exploration of potential efficiency improvement and use of other formalisms and tools remains as significant future work.

It is difficult and costly to develop models, from scratch, that reflects the domain and is enough abstract to be explored exhaustively. It is thus necessary to provide models that can be used commonly for examining physical prerequisites in pervasive computing. Focusing on the specific domain, i.e., physical interactions and their prerequisites, will allow for more sophisticated tool support, as [7] which provides a useful verification tool in the service-oriented computing area.

Event Calculus has been used for specification of policies for event-based system management [3]. It has been recently used for specification of service/agent contracts, such as permissions, obligations, and their changes in response to events [6, 8, 12]. This study explores use of Event Calculus in the pervasive computing area, but does not explore potential usages at runtime, e.g., predicting occurrences of undesired situations as mentioned in [12].

6. SUMMARY

This study discussed a theoretical framework for formal modeling of requirements, assumptions and behaviors for application software in pervasive computing. This study specifically focused on prerequisites for physical interactions, which are defined and examined in terms of scopes and their relationships not limited to tree structures. This study expected the formal modeling to be the basis of analysis and verification, and provided initial experiments using an existing reasoner. We believe this study provides the basis for methodologies and tools for early analysis and verification of application software in pervasive computing, before examining implementation details.

Acknowledgment

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

7. REFERENCES


