Policy-Based Ambiguity Reduction in Pervasive Context-Aware Systems

Sherif G. Aly

The American University in Cairo, Cairo, Egypt

Summary
There has been an ever-increasing interest in context-aware computing expressed by the pervasive computing community. As researchers attempt to create pervasive systems that are unobtrusively embedded in the environment, completely connected, intuitive, effortlessly portable, and constantly available, often do they run into the problem of ambiguity in the determination of the surrounding context. With the presence of context ambiguity, pervasive systems become more incapable of adapting themselves with the surrounding environment. This article describes a policy-based framework for reducing ambiguity in context aware systems. Experimental results show the performance of the system as the number and size of disambiguating policy rules increase.

Key words:
Pervasive Systems, Context Awareness, Ambiguity Reduction, Policies.

Introduction

Context awareness in pervasive computing has been gaining ever increasing research attention. Pervasive systems are required to realize knowledge of their surroundings so as to become better integratable and adaptable to the heterogeneity of their surrounding environment. Much research has been proposed to allow pervasive systems to become context aware. Some researchers propose the integration of context within existing pervasive systems, while others use context acquisition [1], or a combination of both.

As such, a key characteristic for context aware systems is that they must maintain the capability of acquiring and using context related information through interaction with an environment that is sensor-rich, and that is also capable of providing accurate information about itself.

Sensing devices can provide pervasive systems with information such as the location of people and devices [2], however; only an intelligent system would be able to utilize the aggregated data into a meaningful, more useful form, a form that will allow a pervasive system to more naturally interact with users, hence going beyond the legacy of isolated interaction [2].

In most cases, context awareness will involve capturing and making sense of imprecise and sometimes conflicting data and uncertain physical worlds. Various components of a pervasive system must then be able to reason about uncertainty and reduce ambiguity associated with gathered contextual information [3].

Many challenges affect context awareness in general. Such challenges include the representation of context data, the integration of such data with existing systems and applications, the storage and distribution of context data, and the frequency of context retrieval. Furthermore, one of the striking challenges affecting context awareness is the reduction of ambiguity associated with context data.

This article describes a policy-based framework for ambiguity reduction in pervasive systems. The framework can be utilized with other existing ambiguity reduction mechanisms to further reduce contextual ambiguity within a pervasive system.

Users roaming within this system are continuously sensed by a plethora of distributed sensors. Sensors can gather primitive, or raw, data about various users within the system, but however cannot provide high level information such as whether the given user is in a meeting or not, or whether the user is currently evacuating the building. Since users continuously provide raw state information to the system, we label them as providers.

An aggregator periodically, and on-demand uses raw state information of various providers, along with user defined policies stored in a policy repository to generate high-level macro-contextual information using the services of a context engine. The context engine makes use of the user-defined policies, along with raw state information stored in the aggregator, to infer and generate macro-contextual information for various providers of the system. Furthermore, every macro-context is coupled with a calculated level of confidence describing how sure the context engine is of the evaluated macro-context of a given provider. Periodically, the aggregator archives historical macro-contextual information in a context repository. Historical information proves to be very valuable in the futuristic determination of context.

Aggregators within the system can be queried for macro-contextual information. By changing various policies, the

aggregator's interpretation of various collections of raw data, and eventually its inference of ambiguous context, along with the corresponding contextual confidence probability can be altered.
The article shall describe related work, the overall architecture of the system, the various subsystem interactions, along with a detailed description of the aggregator, the context engine, the definition of various policies within the policy repository, and the context repository itself.

2. Related Work

Many contributions have been made towards ambiguity reduction in pervasive systems. Ranaganathan proposes in [3] an uncertainty model based on a predicate representation of contexts and associated confidence values. The model uses various mechanisms such as probabilistic logic, fuzzy logic, and Bayesian networks.
In [7], Dey presented a definition of context, and introduced a conceptual framework to assist in the design of context aware applications. He also attempted to explore some of the challenges associated with implicit context in an attempt to alleviate some context related ambiguity, especially as relates to the inability of sensors to provide accurate sensing data. The author proposed the reduction of ambiguity through one of three approaches, namely (1) to allow applications to know of the inability of sensors to provide accurate information, and hence take appropriate action or (2) to support sensor redundancy for accuracy purposes, or (3) to allow the user to manually remove ambiguity through interaction with a user interface. The third choice was the explored approach by the author, of which the major drawback includes manual user intervention for ambiguity reduction. In [8], he also developed an architecture to support the mediation of ambiguity in recognition-based GUI interfaces. The architecture supported timely delivery and update of ambiguous context, yet still lacking solutions to ambiguity reduction.
In [17], a framework for the analysis, requirements, and design phases of developing context aware systems was developed. In [18], The Context Broker Architecture Ontology was created, which is a very interesting development in context awareness. A broker monitors and controls information used by context aware systems, however, not much contribution was made to reduce ambiguity also that may almost be imminently present in such systems.
In [19], another ontology was developed to allow for better context modeling in pervasive computing environments. The main goal of the work was to classify context types, and give different weight to different types of relevantly important contexts within applications. Context was divided into three types, namely for users, computer entities, and physical entities (such as light, and noise). Policies used predicates to describe context. For example, <Tom status WatchingTV> indicated the obvious state of Tom, namely watching TV. In [20], a Java Context Awareness Framework was developed and included a set of APIs to describe context.
In [9], Chalmers presents how relationships between real world actors and contextual information can be formulated in the presence of uncertainty.
In [10], Thomson identifies some drawbacks associated with situation determination offered by some state of the art context aware infrastructures. Some drawbacks include the use of large logic programs or Bayesian networks, the inability to perform correlations with scaling systems, and the lack of support for ad hoc situation determination. Thomson presents an approach to situation determination that attempts to address such drawbacks, and adds the capability of recognizing ad hoc situations.
In [11], Johnson describes how an architecture for an intelligent environment context supports the changes of representation of knowledge across a range of different programming styles.
In [12], Loke explores argumentation as a reasoning mechanism in context-aware systems, and more expressive rules for user programming of context aware systems.
Grimm presents in [4] and [5] a highly detailed system architecture for pervasive computing that accommodates embracement of contextual change, however, does not fully accommodate context ambiguity. The system however provides an integrated framework for building adaptable applications that allow for user collaboration and devices and applications that easily assimilate together. The idea of policy utilization in pervasive systems on the other hand has also been an issue of research as presented in [14-16].
Although the contributions mentioned above propose tentative solutions for the difficult and challenging context ambiguity problem, our framework is not particular for specific pervasive application like many other solutions are, and avoids the need for complicated fuzzy logic or Bayesian networks. Furthermore, the framework relies on user defined policies that allow the system to reduce ambiguity associated with context related information to achieve more abstract and high-level context. The framework can be utilized with existing context ambiguity approaches to further contribute in ambiguity reduction.

3. System Architecture

At a high level, our system is composed of the ability to gather context information from those entities that generate it, store context, interpret basic context into more intelligent context, and archive context. In order to achieve such functionalities, we created seven components in our policy-based ambiguity reduction system, as shown in Figure 1. The system is composed of:

- **The Providers:** The users who feed their information to the system through the sensors.
- **The Sensors:** To collect information about providers such as location sensors, voice sensors, movements sensors, etc.
- **The Aggregators:** To collect and retain up-to-date information about the providers from the sensors, to induce the context engine for generating higher level context information, and to answer queries about context information.
- **The Elicitors:** Those users that query for context related information about certain other users.
- **The Context Repository:** To archive historical context information.
- **The Policy Repository:** To store policies directing the inference of raw and low-level context information into higher level context-information.
- **The Context Engine:** Uses up-to-date user information in the aggregators, along with policies stored in the policy repository to generate higher level contextual information.

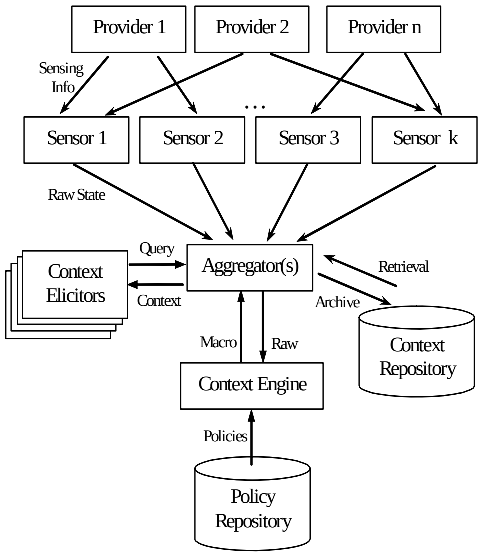

Figure 1 System Architecture

The architecture of the system primarily relies on the presence of one or more aggregators that constitute the focal point of distribution of macro (high level) contextual information of various providers in the system. A typical scenario for information flow starts with providers, namely users, that feed low-level information into the system via sensors. Depending on the type of the sensor itself, the proper low level information (raw information) will be gathered. For example, motion sensors detect motion information; sound sensors detect the presence of sound, proximity sensors, detect proximity, and so on. Various such sensors will obtain very specific raw context information about providers, and then relay them to an existing aggregator.

It is up to the aggregator to rely on a predicate-based context engine to convert the very specific context information obtained from various sensors, such as provider locations, sound and motion detection, and proximity detection into a more abstracted and high level provider-related context such as the provider's existence in a meeting, or its evacuation of a building during a fire drill, or its delivery of a presentation to executives of the organization. We call such high level context a macro context.

The context engine uses the raw provider states stored in the aggregator, along with user defined policies stored in a policy repository to infer macro contexts. The inferred macro contexts are then stored back again in the aggregator to add to a process of continued learning.

Eventually, both the raw context information, and the inferred macro-context information become both significant in quantity and obsolete. Periodically, the aggregator archives contexts into the context repository. Such archived contexts can be used by the context engine to obtain a historical insight into previously computed provider contexts. The context engine can use such historical context information, with reliance on basic temporal and spatial locality principles to support its context derivation. For example, the provider's attendance of a meeting regularly for the past month during a specific time is a very good indicator that the provider will continue to attend such meetings, probably at the same time in the near future.

Eventually, the gathering of low-level context information, and the inference of macro-contextual information is only useful if it can be used. Context elicitors are entities interested in obtaining context-related information about various providers. An elicitor may be a provider itself, or any other entity. Not only can elicitors obtain macro contextual information, but they can also receive lower level context information about providers, such as a sensor detecting the proximity of a user.

4. The Providers

Every context-aware system is composed of various sensors of different types. In [13], sensors are divided into physical, virtual, and logical sensors based on the functionality of each. However, the most obvious examples of sensors are those physical sensors that are capable of providing information about various physical entities within the system.
Examples of physical sensors include location sensors, proximity sensors, biometric sensors, and magnetic card readers. The author in [13] also continues to describe virtual sensors as those sensing information from virtual worlds such as networks, or operating systems, and logical sensors that infer information from both physical and logical sensors.
In this system, we introduce the notion of a provider. A provider is any entity providing sensors with information. The provider itself could be either physical or logical. Examples of physical providers include humans walking down a corridor and being sensed by proximity sensors, or the same humans swiping their badge into a magnetic card reader, or even performing a retinal scan to access a secured environment. On the other hand, logical providers can include operating systems sending a system overload alert, or a network providing its traffic status. In either case, providers, whether physical or logical, provide various sensors with information about their status.
As such, any sensor, irrespective of the type of the sensor, detecting information associated with a given provider will then generate a raw state for the provider. The raw state, as shown in Figure 2, will minimally consist of a timestamp, an identification of the sensed provider, and a predicate describing both who was detected and the type of detection itself. The predicate itself is described later. The raw state information about the provider is then immediately registered at the aggregator.

Time Stamp	Sensor ID	Context Predicate

Figure 2 Generated Raw State

Given many such providers in a system detected by different kinds of sensors, the aggregator will always contain up to date low level information about the existing providers. Figure 3 illustrates the data flow of how state information of some provider is generated and relayed to an aggregator. The provider walking down a hallway for example is detected by a proximity sensor. As a result of such detection, the proximity sensor will generate raw state information associated with the detected employee, and will relay such raw information to the aggregator. The aggregator will then use, at a later stage, such information to generate macro contextual information.

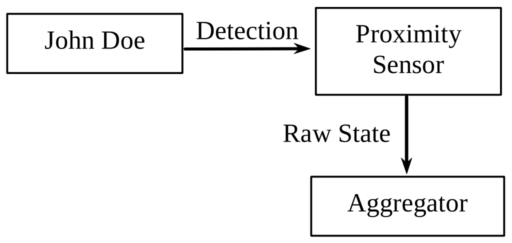

Figure 3 Raw State Aggregation

5. The Predicates

The usage of predicates in this system is of great significance. As previously stated, a raw state propagated from an existing sensor and registered at an aggregator contains as a minimum, a timestamp, an identification of the sensed provider, along with a predicate describing the state. The usage of predicates provides a simple and uniform representation for different kinds of raw contexts that can then be easily used by the context engine to generate macro contextual information.

A predicate is of the form:

(<Subject>, <Verb>, <Object>)

The subject above is the identifier of the provider, and the object is the identifier of the sensor detecting an action related to the subject in one way or another. If the provider cannot be identified, the subject is simply “null”. The type of action detection is dependent on the type of the sensor. Motion sensors detect motion actions; sound sensors detect sound related actions, and so on. The verb on the other hand describes the sensing activity itself. Currently, five common types of actions to be monitored are identified, namely related to:

- Motion detection.
- Sound detection.
- Proximity detection.

- Location detection.
- Pressure application detection.
- Pressure release detection.

Table 1 illustrates the various actions, and their corresponding verbs to be used in the predicates.

Table 1: Supported Actions and Corresponding Verbs

Action Type	Predicate Verb
Motion Detection	*MoveBy*
Sound Detection	*HeardBy*
Proximity Detection	*CloseTo*
Location Detection	*In*
Pressure Application Detection	*Pressured*
Pressure Release Detection	*Released*

As an example, a provider identified as John and detected by a proximity sensor located in the Room1 will have the following predicate generated by the proximity sensor:

(John, CloseTo, Room1)

The subject is the provider identification, the verb is highly dependent on the sensor type, which happens to be a proximity detector in this case, and the object signifies the sensor name, and with proper naming, one can signify what the proximity detector references. As another example, a location detector, which is different than a proximity detector, detecting the exact presence of John in Room1 will have a predicate generated as follows:

(John, In, Room1)

Of course, detecting a location of a user in itself may be performed by a logical sensor, since location detection may require the presence of multiple sensors in itself. The same applies to other types of sensors also.

The usage of a general and simple form for a predicate as indicated above provides an easier processing of the predicates themselves, along with providing a uniform representation of predicates, independent of the various sensors.

6. The Aggregator

As indicated earlier, the aggregator(s) constitute the focal point of distribution of macro-contextual information of various providers in the system. Various sensors in the network continuously forward raw state information about providers to the aggregator. The aggregator will then add the raw states to its collection.

A context engine assists the aggregator in generating macro-contextual information about various providers from the set of raw states available at the aggregator itself. The context engine uses user-defined policies that indicate how the transformation should happen from a set of raw states to a higher level representation of provider context with a given confidence level. The context engine will be explained later in this article.

At any point in time, a context elicitor can query the aggregator to obtain either raw context information or macro contextual information about a given provider or providers. Macro-contextual information about providers contain a given confidence level that indicates to the elicitor to what extent the aggregator believes that this is the proper context of the provider.

As an example, raw states of a given provider may indicate that the provider has entered a meeting room, that the provider is present along with two other people in the same room, that a sound is detected from the meeting room, and that a projector is activated. According to a general form of a policy set by the user, the aggregator can conclude with the help of the context engine that the given provider is in a meeting, and with a given confidence level.

As raw states and calculated macro-contextual information is stored in the aggregator, and as this information becomes irrelevant to the immediate time, the aggregator will need to archive such information. Periodically, the aggregator will archive raw states, as well as generated macro-context states in the context repository.

7. Policies and the Policy Repository

Policies reside in the policy repository. Such repository is continuously queried by the context engine to allow the engine to determine how to compute macro-contexts. Context related information is then sent back to the aggregator, thus signifying the latest up to date macro-context of a given provider.

7.1 The Need for Policies – an Example

As an example, a user, which we call a provider here, roams around a building and is continuously detected by various types of sensors distributed around the building. Raw information about the user is therefore collected by various sensors within the building. The objective is to create a policy capable of identifying a macro-context associated with the provider. In other words, high level questions of the following type need to be answered: Is the provider currently in a meeting? Is the user evacuating the building during a fire drill? Is the user on lunch break? Not only should the system provide answers to such questions with an affirmation or negation, but should rather indicate a confidence level associated with its answer. In other words, the provider is in a meeting with a confidence level of 0.9, where the maximum confidence level is a 1.

The definition of various policies directs the context engine as to how to generate answers for such questions such as the ones indicated above. As an example, a meeting policy is created that states the following:

- If three or more proximity sensors around a specific meeting room detect the presence of the provider.
- If one or more different other providers are detected also in the same meeting room.
- If one or more audio sensors surrounding the meeting room detect noise, or more intelligently at a later stage, a human conversation.
- Then according to such policy, it can be concluded that the provider is in a meeting at a specific location identified by the various sensor locations.

However, determining the presence in a meeting is not sufficient; the question to be asked further is, with what level of confidence is the user in a meeting? Various confidence levels are thus associated with each component of the policy. All confidence levels must sum up to a system defined maximum of 1. Table 2 below shows the association of various confidence contributions to the example indicated above. The detection of proximity surrounding the meeting room itself contributes with a confidence level of 0.5. The detection of other providers in the same room contributes to a confidence level of 0.3, and the detection of noise contributes to a confidence level of 0.2. All of the contributions should add up to a system defined maximum of 1.

Table 2: Policy Confidence Level Contributions

Policy Component	Confidence Contribution
>=3 Proximity Sensor Detections	*0.5*
>=1 Association	*0.3*
>=1 Audio Detection	*0.2*
$\sum$	***1***

If no confidence levels are specified, the various components will assume equal contributions to the overall evaluation of confidence. If only part of the confidence contributions are specified, and the rest are not, the unspecified contributions will assume equal values of whatever remains to achieve a maximum contribution of 1 by all policy components.

7.2 Policy Representation

As shown in Figure 4, policies stored in the policy repository are represented using exactly one context node and one ore more policy nodes. A context node identifies the type of macro-context to be computed such as a meeting, while the policy nodes determine the various components contributing to the evaluation of the macro context itself. Each policy node in itself may be either simple or compound. A compound policy node may require the evaluation of another macro-context, and thus as shown in the figure below, the policy node may be linked to another context node with its associated policy nodes. A context node and its policy nodes are represented using linked lists.

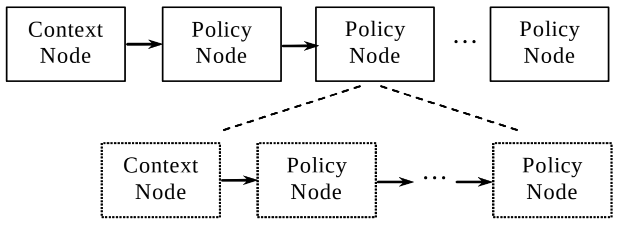

Figure 4 Policy Representation

The policy repository itself may be composed of multitudes of user defined policies as such. Each policy will have its usual context node and policy node(s) representation to store information associated with the policy as shown in Figure 5.

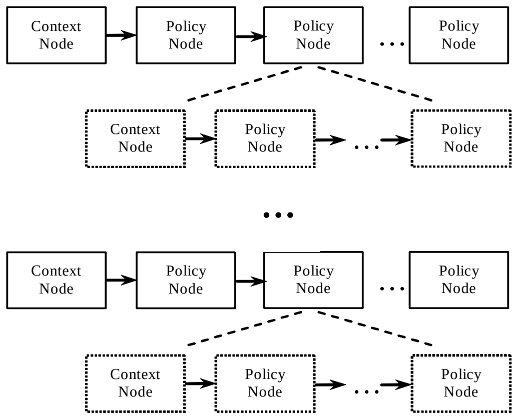

Figure 5 Policy Repository

Figure 6 shows the internal structure of a context node, and the corresponding policy node. A context node is composed of a name identifying the macro-context, while each policy node is composed of an action type, an occurrence, a relation, and a weight. Each context node is associated with multiple policy nodes.

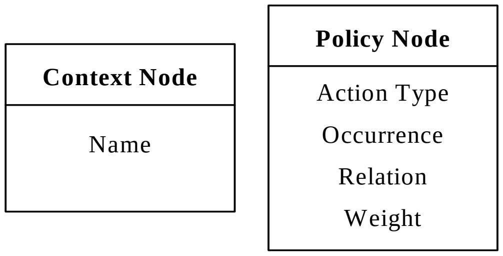

Figure 6 Context and Policy Nodes

The action type field: such as a “proximity detection”, or “audio detection”, signifies the type of sensor detection required for this policy component to be valid. For example, a proximity detection action type indicates that proximity detection is required for this macro-contextual evaluation.

The occurrence field: is a numeric value indicating the quantity of action types required for the evaluation to be valid. An occurrence value of three for example, and an action type of “proximity detection” does not necessarily mean that exactly three proximity detections are required, but rather, it is up to the relation field to determine whether we need exactly three, more than three, more than or equal to three, less than three, or less than or equal to three.

The relation field: contains one of five relational operators: <, <=, >, >=, or == to determine along with the occurrence field how many occurrences are needed for this policy rule to be satisfied.

The weight field: indicates the contribution of this policy node to the overall evaluation of the macro context. For example, proximity detection as indicated in this policy node may contribute 0.5 to the overall evaluation of this context.

7.3 Policy Example

As an example shown in Figure 7, the determination whether a provider is in a meeting or not, and as indicated earlier, may, according to this example be determined according to the following weights (that sum up to one):

- If three or more proximity sensors around a specific meeting room detect the presence of the provider, with a weight of 0.5.
- If one or more different other providers are detected also in the same meeting room, with a weight of 0.3.
- If one or more audio sensors surrounding the meeting room detect noise, with a weight of 0.2.

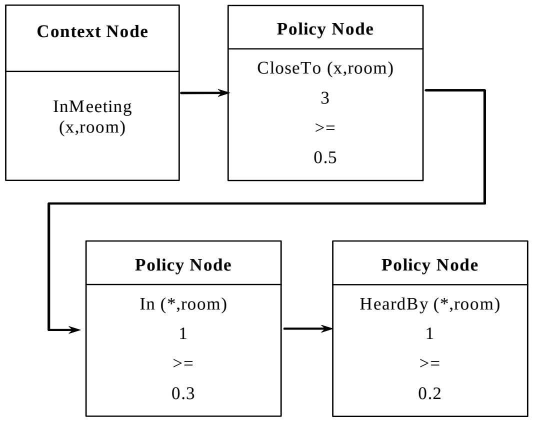

Figure 7 Context and Policy Node Example

As shown in Figure 7 above, the context and policy node construction is composed of a single context node with a name “InMeeting”, which operates on some provider “x”, and location “room”. The purpose of this context node is to show the policies determining whether provider “x” is in a meeting in a location “room”.

According to the example in Figure 7, there are three policy nodes created. Each policy node contains an “Action Type”, an “Occurrence”, a “Relation”, and a “Weight”. The first policy node identifies the first rule of the example, namely whether three or more proximity sensors around a specific meeting room detect the presence of the provider, with a weight of 0.5. The “Action Type” for this policy node is populated with “CloseTo(x,room)”, the “Occurrence” is populated with “3”, the “Relation” is populated with “>=”, and the weight is populated with “0.5”, thus indicating that more than three proximity detection should be found for x, and that this policy in itself contributes 0.5 of the overall weight.

Similarly, the two other policy nodes are created with “Action Types” “In” and “HeardBy” to represent the remaining other two rules. The “*” in those policy nodes indicates the detection of “any provider”. For example, “In(*,room)” indicates the detection of “any provider” in the given "room" with an occurrence of >=1, and a contribution of 0.3.

7.4 Compound Policy Nodes

Compound policy nodes are those nodes whose evaluation depends on the evaluation of other policy nodes, either compound or simple. The support for compound nodes within this system facilitates the definition of policies and allows for a higher level of abstraction when defining such policies.

8. The Context Engine

The context engine within our system is the sole entity containing the logic capable of converting raw context information about a given provider into macro-contextual information. The context engine's services are only available to the aggregator coupled with such context engine. When a context elicitor first issues a query to the aggregator asking a macro question about a given provider such as: Is the given provider in a meeting? Is the provider currently having lunch? Or a general question such as: What is the given provider currently doing? The query request is forwarded by the aggregator to the context engine coupled with such aggregator. The aggregator then utilizes the services of the context engine to answer such question.
As a prerequisite, the context engine requires the presence of policy rules, along with the raw state information in order to come up with conclusive decisions regarding the answer to the submitted query. We can subdivide queries into two major types namely specific queries, and general queries as indicated above.

8.1 Specific Queries

Specific queries are usually simpler to answer. A specific query as indicated earlier could be of the form: Is a given provider currently in a meeting? When the context engine receives a query as such, the policy rule evaluation domain is significantly reduced. Query examples are indicated below:

- The format of a query issued to check whether a provider identified as John is in a meeting in any room is of the following form:

  InMeeting(John,*).

- If the query is intended to check if John is in a meeting in room1, the query if of the following form:

  InMeeting(John, Room1)

In such case, it is obvious that the context node related to meetings should be traversed, its policy rules evaluated for the given provider, and the question about whether the given provider is in a meeting or not answered along with a calculated level of confidence.
As shown in the flowchart in Figure 8, the context engine initially starts by locating the context node of the relevant policy rule related to the query being submitted. In this example, the query is related to presence in a meeting. If the relevant context node of the policy rule is found, the context engine is immediately capable of answering the question with a given level of confidence, given that corresponding raw state information for such provider exists. If the rule is not present however, the context engine is simply incapacitated from answering such query. The presence of policy rules within the policy repository is therefore crucial and a prerequisite for being able to answer such queries.
After locating the context node for a meeting within the policy repository, the context engine then starts evaluating every single policy node associated with the context node. The action type of the policy node indicates to the context engine what kind of raw states to look for at the aggregator. If the action type is proximity detection, the context engine will attempt to locate proximity detections for the given provider being queried. The occurrence field along with the relation field indicates to the context engine how many such proximity detections for the given provider to look for. If the relation and occurrence in our example are ">" and "3", this means that the context engine is looking for more than three occurrences of proximity detections. Once more than three are found, the context engine can safely stop evaluating this specific policy node. Furthermore, if the condition is also met, then the weight contribution of this policy node to the overall context evaluation is taken into consideration. If the condition is not met, the weight contribution of the policy node currently being evaluated is not taken into the overall evaluation of the context. The summation of weight contributions of all satisfied policy nodes is the overall confidence level. The weight contribution of each policy node thus signifies to a great extent how important that policy node is to the overall evaluation of the queried for context.

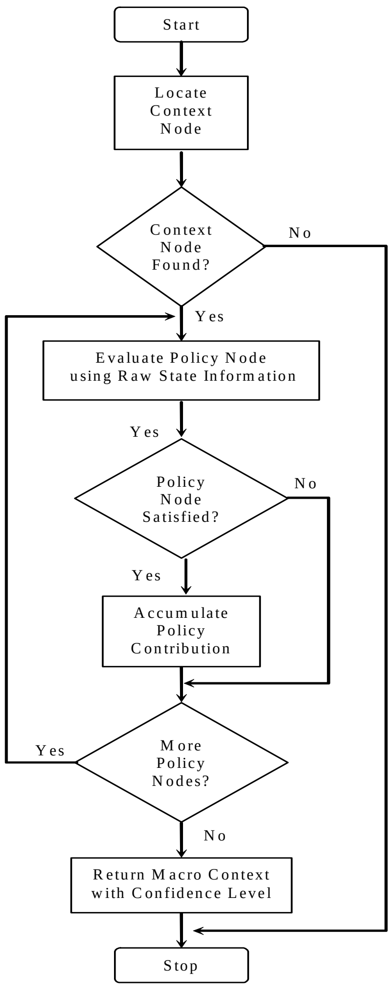
Start
Locate Context Node
Context Node Found?
No
Yes
Evaluate Policy Node using Raw State Information
Yes
Policy Node Satisfied?
No
Yes
Accumulate Policy Contribution
Yes
More Policy Nodes?
No
Return Macro Context with Confidence Level
Stop

Figure 8 Evaluating a Specific Query

8.2 General Queries

General queries on the other hand are relatively more demanding to evaluate. To answer a general question about the current context of a given provider requires an exhaustive evaluation of all relevant context nodes. Each context node is traversed, and its corresponding policy nodes evaluated. Eventually, each context node will have a corresponding level of confidence associated with it depending on the outcome of the evaluation of policy nodes within the context node itself. The contexts, along with their levels of confidence are then returned back as part of the query response itself.
For example, given three policy rules are populated in a policy repository, namely for meetings, lunch, and evacuation. Given also that raw state information is gathered at the aggregator, a general query asking about the current context of a provider John Smith will return back with an output that looks similar to the following show in Figure 9:

John Smith is Currently

In Meeting	Confidence 95%
Evacuating	Confidence 60%
At Lunch	Confidence 30%

Figure 9 Sample Output of a General Query

The services of the context engine are only available to an aggregator. Figure 10 below shows how a context elicitor first requests a macro-context from an aggregator, and how the aggregator invokes the context engine. The context engine on the other hand uses the data stored in both the aggregator (in the form of raw states), and that stored in the policy repository (in the form of policies) to evaluate the query.

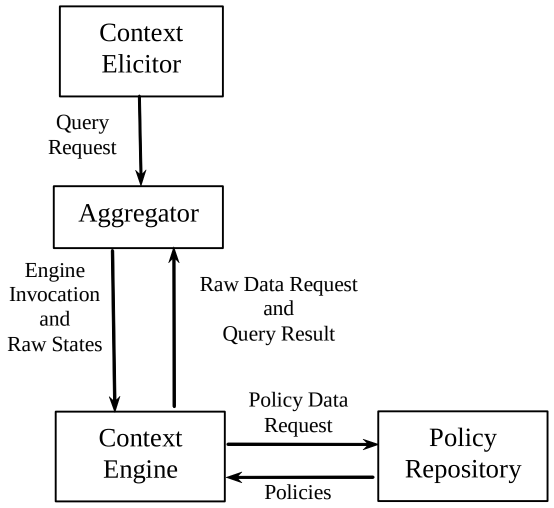

Figure 10 Interactions for Query Answering

9. Experimental Results

We conducted implementation and experiments to determine the performance behavior of the system as policy rules became more complicated. As indicated earlier, a specific query submitted to the system will entail the evaluation of a single policy rule. Such policy rule is composed of a single context node describing the rule itself, and a number of policy nodes.

We measured the amount of time taken to fully evaluate a policy rule as we increased the number of policy nodes within the rule itself. Compound policy nodes are composed of a number of policy nodes also themselves. We started off with a single policy rule composed of one single policy node. We varied the number of policy nodes from one to thirty, each time adding a policy node to the already existing policy nodes. With each run, we measured the total time taken to evaluate the entire rule itself given the number of policy nodes composing that rule. We were not only interested in the amount of time taken to evaluate the rule itself, but rather the behavior of the evaluation time as the number of policy nodes increased.

The results obtained in Figure 11 illustrate the total time taken by the system to evaluate the rule as the number of policy nodes ranges from one to thirty. The graph below illustrates that as the number of policy nodes increases, the rule evaluation time increases gracefully in a semi-linear fashion, that, which makes the system gracefully scalable in terms of rule size. Of course, the exact evaluation time highly depends on the type of policy nodes within the rule itself, but the semi-linear behavior of the system is illustrated below.

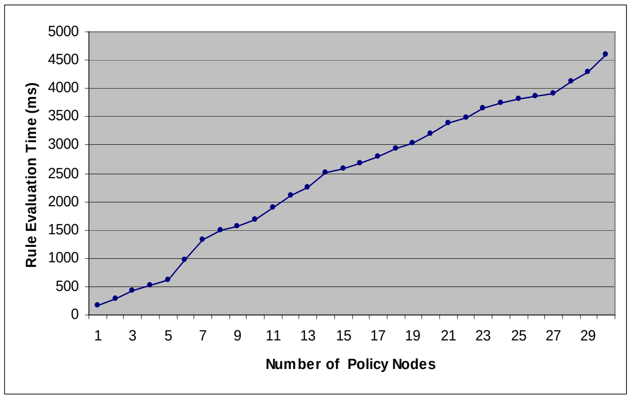

Figure 11 Behavior of Rule Evaluation Time

Given the behavior of the system in evaluating specific queries as indicated in the figure above, more general queries will entail the evaluation of multiple policy rules. Figure 12 below shows the behavior of the system as multiple rules are evaluated to answer a given query. The same experiment as above was conducted, yet with increasing the total number of rules to be evaluated. The same semi-linear behavior of the system is exhibited, yet with an increased rule evaluation time proportional to the number of rules being evaluated.

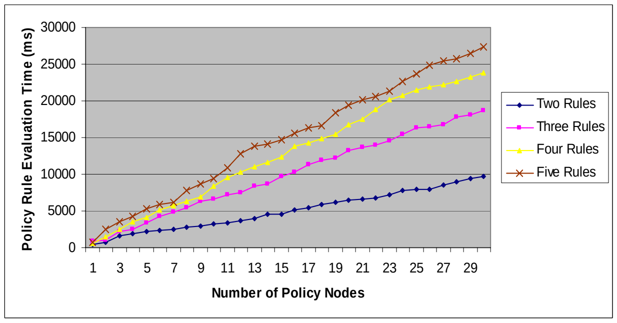

Figure 12 Behavior of Multiple Rule Evaluation Time

10. Conclusion

Capturing context-related information poses itself as one of the fundamental aspects of building useful pervasive systems. In many scenarios, context-related information is either ambiguous or conflicting. In this article, we presented a policy-based framework to reduce ambiguity in context aware systems. Multitudes of sensors continuously capture the state of various providers roaming around a given environment, and generated raw states are forwarded to an aggregator. Well-defined policy rules, in the form of predicates, defining the interpretation of raw states are stored in a policy repository. Context elicitors can query about the status of various providers within the system, after which the aggregator invokes the services of a context engine to answer questions regarding the providers in question. The context engine utilizes the raw states at the aggregator, along with the various defined policy rules to infer a macro-context of the providers in question. Such macro-context is coupled with a confidence level indicating how certain the aggregator is in its context inference. Eventually, both raw-contexts and macro-contexts are invalidated and archived into the policy repository. Conducted experiments demonstrated the performance of the system as the number of policy nodes increased, and as the number of evaluated rules also increased. Our future work involves the utilization of historical context related information for increased ambiguity reduction.

References

[1] G. K. Mostefaoui, J. Pasquier-Rocha, and P. Brezillon , “Context Aware Computing: A Guide for the Pervasive Computing Community”, The IEEE/ACS International Conference on Pervasive Services (ICPS’04), Lebanon, 2004.

[2] D. Saha and A. Mukherjee, “Pervasive Computing: A Paradigm for the 21st Century” *IEEE Computer*. Pp. 25-31, March 2003.

[3] A. Ranganathan, J. Al-Muhtadi, and R. Campbell, “Reasoning about Uncertain Contexts in Pervasive Computing Environments” *IEEE Pervasive Computing*, Pp. 62-70, April-June 2004.
[4] R. Grimm, “One.World: Experiences with a Pervasive Computing Architecture”, *IEEE Pervasive Computing*, Pp. 22- 30, July-September 2004.
[5] R. Grimm, “System Support for Pervasive Applications”, *ACM Transactions on Computer Systems*, Pp. 421-486, November 2004.
[6] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. Campbell, and K. Nahrstedt, “A Middleware Infrastructure for Active Spaces,” *IEEE Pervaisve Computing*, Pp. 74-83, 2002.
[7] A. K. Dey, D. Salber, and G. D. Abowd, “A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications”, *Human-Computer Interaction Journal*, Pp. 97-166, 2001.
[8] A. Dey, J. Mankoff, G. Abowd, and S. Carter, “Distributed Mediation of Ambiguous Context in Aware Environments”, The Eighteenth Annual ACM Symposium on User Interface Software and Technology, Paris, France, 2002.
[9] D. Chalmers, N. Dulay, and M. Sloman, “Towards Reasoning about Context in the Presence of Uncertainty”, The Workshop on Advanced Context Modeling, Reasoning, and Management, United Kingdom, 2004.
[10] G. Thomson, P. Nixon, and S. Terzis, “Towards Ad-hoc Situation Determination”, The Workshop on Advanced Context Modeling, Reasoning and Management, United Kingdom, 2004.
[11] C.W. Johnson, K. Carmichael, and H. Kummerfeld, “Context Evidence and Location Authority: the Disciplined Management of Sensor Data into Context Models”, UbiComp 2004, Nottingham, England, 2004.
[12] S. Loke, “Facing Uncertainty and Consequence in Context-Aware Systems: towards an Argumentation Approach”, CW2004, Tokyo, Japan, 2004.
[13] J. Indulska and P. Sutton, “Location Management in Pervasive Systems”, The Workshop on Wearable, Invisible, Context-Aware, Ambient, Pervasive, and Ubiquitous Computing, Australia, 2003.
[14] A. Patwardhan, V. Korolev, L. Kagal, and A. Joshi, “Enforcing Policies in Pervasive Environments”, The IEEE International Conference on Mobile and Ubiquitous Systems Mobiquitous, Boston, Massachusetts, 2004.
[15] L. Kagal et. Al, “A Policy Language for a Pervasive Computing Environment”, The IEEE International Workshop on Policies for Distributed Systems and Networks, Italy, 2003.
[16] G. Tonti, J. Bradshaw, R. Jeffers, R. Montanari, N. Suri, and A. Uszok, “Semantic Web Languages for Policy Representation and Reasoning: A Comparison of KAoS, Rei, and Ponder”, The International Semantic Web Conference, Florida, 2003.
[17] S. Loke, and E. Syukur, “The MHS Methodology: Analysis and Design for Context-Aware Systems”, The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, Korea, 2006.
[18] H. Chen, “An Intelligent Broker Architecture for Context-Aware Systems”. Ph.D. Dissertation, University of Maryland Baltimore County, 2004.
[19] X. Lin, S. Li, Z. Yang, and W. Shi, “Application-Oriented Context Modeling and Reasoning in Pervasive Computing”. The Fifth IEEE International Conference on Computer and Information Technology, New York, 2005.
[20] J. Bardram, “The Java Context Awareness Framework (JCAF) – A Service Infrastructure and Programming Framework for Context-Aware Applications”, The Third International Conference on Pervasive Computing, Germany, 2005.

Sherif G. Aly received his B.S. degree in Computer Science from the American University in Cairo, Egypt, in 1996. He then received his M.S. and Doctor of Science degrees in Computer Science from the George Washington University in 1998 and 2000 respectively. He worked for IBM during 1996, and later taught at the George Washington University from 1997 to 2000 where he was nominated for the Trachtenberg prize-teaching award for his current scholarship and scholarly debate. He spent two years as a guest researcher for the National Institute of Standards and Technology at Gaithersburg, Maryland from 1998 to 2000. Dr. Aly also worked as a research scientist at Telcordia Technologies in Morristown, New Jersey, in the field of Internet Service Management Research, and as a Senior Member of Technical Staff at General Dynamics Network Systems. He also consulted for Mentor Graphics and taught at the German University in Cairo. He is currently a faculty member at the Department of Computer Science at the American University in Cairo. Dr. Aly published numerous papers in the area of distributed systems, multimedia, digital design, and programming languages. His current research interests include pervasive systems, programming languages, multimedia, directory enabled networks, and image processing. Dr. Aly is a member of IEEE.