Interpretation of Inconsistencies via Context Consistency Diagrams

Viktoriya Degeler and Alexander Lazovik
Distributed Systems Group, Johann Bernoulli Institute
University of Groningen
Nijenborgh 9, 9747 AG, The Netherlands
{*V.Degeler,A.Lazovik*}*@rug.nl*

***Abstract*—Pervasive context-aware systems base their responses on information about the environment collected from ubiquitous sensors. The inevitable drawback of such systems is that raw data collected from sensors is often noisy, corrupted, and imprecise. Erroneous sensor readings create uncertainties and ambiguous interpretations. Thus creating an interpretation challenge for the context-aware system that needs to reason about possible states of only partially observable subjects.**

We propose a mechanism for pervasive context-aware systems to process the information gathered from sensors so to obtain knowledge about possible environment states. This includes both the ability to reason about a situation with incomplete knowledge and to cope with erroneous contexts. We present a probabilistic approach to reason about the likelihood of each particular situation, state of a variable, and variable interdependence. The evaluation shows that the proposed approach is applicable to real-time context inference problems.

***Keywords*-Context-aware computing, context reasoning, context inconsistencies.**

I. Introduction

Pervasive context-aware systems gather information from the environment to adapt application behavior without an explicit user intervention. A typical example is a smart home system [1] that monitors needs of users and coordinates low-level tasks of home devices. Such a system is responsible for the following three activities: (i) raw data collection from the sensors; (ii) data transfer to the system middleware that processes the acquired data and draws logical conclusions from it, that is, it creates consistent interpretations of the environment; (iii) high-level applications use of this information to make appropriate decisions using the information provided by the pervasive system middleware.

The inevitable challenge is the processing of raw data collected from sensors that is often noisy, corrupted, imprecise, and easily leads to inconsistencies. Inconsistencies may be caused by different factors. A study by Jeffery et al. [2] shows that in dynamic environments the percentage of RFID tags reads can drop down to 60-70%. The data may become obsolete rapidly. Due to the asynchronous nature of sensor readings, the data being correct at the time of sensing may turn obsolete at the time of delivery at the end-point applications. The order of raw data arrival may be different from the order of sensing, which may influence the interpretation of the context. Finally, context reasoning algorithms themselves may introduce errors.

In this paper, we propose a mechanism of reasoning about sensor information to define possible context interpretations. This includes both the ability to reason about a context with incomplete knowledge, as well as the ability to cope with erroneous contexts that may lead to false beliefs. We propose a data structure called *context consistency diagrams (CCD)* for efficient tracking of sensed contexts, which can be efficiently maintained and queried in real-time, and used to obtain information about the likelihood of a particular context interpretation, variable or relations between variables.

The rest of the paper is organized as follows. Section II overviews the state of the art in context reasoning. In Section III, we discuss types of context inconsistencies and ways to resolve them. In Section IV, we introduce CCD. Section V shows how CCD can be used to query context information. In Section VI, we show how to maintain CCDs in real-time. We evaluate the approach in Section VII and provide concluding remarks in Section VIII.

II. Related work

Various authors address the issue of inconsistent sensor readings and precise context determination.

Xu and Cheung et al. [3] study the detection of context contradictions based on predefined constraints. They propose to convert each constraint into a tree with constraint operators as vertices and contexts as edges. Then they introduce a partial constraint checking (PCC) algorithm that is capable of re-checking only parts of the constraint tree that may be affected by a new context. Huang et al. [4] propose to check branches probabilistically. This enables fast processing of large trees and adds scalability to PCC, but reduces the percentage of correctly found inconsistencies. The papers aim at fast detection of contradictions, while in the present approach we concentrate on the problem of different context interpretations after inconsistencies are found. Their findings can also be combined with our approach, as inconsistencies found through their method may be interpreted using CCD.

Bu et al. [5], [6] perform context reasoning by modelling the context ontology and then finding inconsistencies using ontological reasoning. They propose to discard conflicting contexts based on their relative frequencies. Xu et al. [7]

propose similar resolution strategies, among which are drop-latest, drop-all, drop-random, and drop-bad. The latter heuristic counts the number of conflicts for each context and drops the one with the biggest number. While those techniques can be used to successfully resolve the straightforward inconsistencies, our approach is helpful when proposed heuristics cannot confidently resolve the conflict, which may lead to retaining the incorrect interpretation.

Henricksen and Indulska [8] classify context properties and outline initial ideas on handling several inconsistencies. While introducing a classification, they do not provide precise algorithms for dealing with context inconsistencies. Lu et al. [9] provide a mechanism for detecting failures in context-aware applications and means to test such applications. Huang et al. [10] study the detection of inconsistencies that emerge due to asynchronous arrival of concurrent events. The proposed algorithm detects the original order of such events based on *happen-before* relation. On the other hand, we do not consider inconsistencies caused by the order in which context information is arriving.

Kong et al. [11] propose to extend the OWL ontology with fuzzy membership to tolerate inconsistencies. Their proposal involves manual assignment of membership values and does not propose a way to retrieve useful information from it.

A similar fuzzy approach to ours is discussed in [12]. The authors try to minimize the impact of early incorrect decisions made during software design. They show that wrong classification of an entity to one of the mutually exclusive classes if done early, may lead to further incorrect or suboptimal design of the software system. They propose to improve the process by deferring decisions about entity's classification as long as possible, instead of assigning fuzzy membership values to each of possible classes. However, the solution is not applicable to context reasoning, as it is based on human decisions about entity's properties membership values that have to be updated with each information change. This is acceptable for the prolonged and slow software development process, but impossible in highly dynamic automated context-aware systems.

III. Architecture

Context-aware reasoning systems produce an application-friendly interpretation of raw sensor data. A possible architecture for such systems is shown in Figure 1. Often, an optional rule-based pre-processing of raw sensor data is performed. In Figure 1, a rule ($TVsound = max \implies TVchannel \in \{sports, shows\}$) is applied to a sensed value ($TVsound = max$). The resulting pre-processed context ($TVsound = max, TVchannel \in \{sports, shows\}$) is then passed to a context subsystem ("CCD representation" layer, see Section IV) that is responsible for efficient storage of acquired context information, resolving inconsistencies, answering to queries, or triggering events to the subscribed top-level applications.

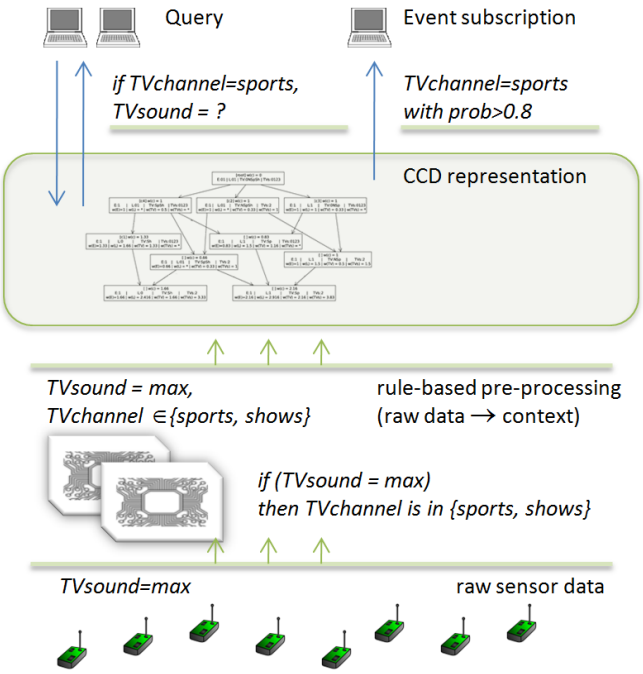


Figure 1: Context reasoning using CCD.

To deal with inconsistencies, we introduce *context consistency diagrams (CCD)*. CCD allow efficient representation of acquired context information together with all possible context inconsistencies and interpretations. CCD is *inconsistent*, if there is no single interpretation that is confirmed by all sensed (and then pre-processed) data. Ideally, conflicts caused by a failed sensor or by data expiration should not stop the system from providing the best possible interpretation for the acquired context information. Several techniques exist to resolve an ambiguous conflict in favor of one interpretation. But if the resolution is incorrect, further interpretations of a situation will also be wrong, even if further information may show that another solution was preferable. To deal with this, a CCD keeps several interpretations, each with its own probability of being true.

We associate a likelihood of being true to each acquired chunk of data. Whenever chunks "support" each other (there is an interpretation of a situation that is consistent with all of them), their mutual truth likelihood is higher comparing to the conflicting ones. Additionally, each arrived and pre-processed information shares a certain degree of truth likelihood, thus compensating the effect of a faulty sensor over the inferred information received from that particular sensor. The most probable interpretation is then the one that is "supported" by the majority of consistent contexts. Even if a particular context does not support the most "popular" interpretation, it is still stored in a CCD. It might happen that with the acquisition of new data, the context is considered more likely, if the new data support it. With such a structure

Table I: Example of environment and context creation.

(a) Variables.

Variable	Domain
Electricity	off, on
Light	off, on
TV	off, news, sports, shows
TV sound	0, 1, 2, 3

(b) Dependency rules.

$\neg(E = off \wedge (L = on \vee \neg(TV = off)))$	Light and TV can be turned on only if electricity is on.
$\neg(TV = off \wedge TVs \in \{1, 2, 3\})$	Non-silent TV sound means TV is turned on.
$TV = shows \Rightarrow L = off$	If TV channel is 'shows', light should be turned off.

(c) Sensor readings and contexts.

ID	Sensor reading	Context
c_1	$TV = Sh$	$E:1 \mid L:0 \mid TV:Sh \mid TVs:0123$
c_2	$TVs = 2$	$E:1 \mid L:01 \mid TV:NSpSh \mid TVs:2$
c_3	$L = 1$	$E:1 \mid L:1 \mid TV:0NSp \mid TVs:0123$
c_4	$TV \in \{Sp, Sh\}$	$E:1 \mid L:01 \mid TV:SpSh \mid TVs:0123$

the context interpretation is never final, as new data may change the interpretation by contributing to an interpretation previously considered wrong.

IV. Context consistency diagram

A server that collects raw data (pre-processing layer in Figure 1) obtains information from the underlying layer in a form $v_i = d_{ij}$, i.e., a variable v_i has a value d_{ij}. It is possible that the sensors return a set of values, i.e., $v_i \in \{d_{ij_1}, d_{ij_2}\}$. For example, a location variable may be sensed by a sensor (e.g. RFID) that is known to be imprecise.

Definition 1 (Environment). *An* environment $\langle V, D\rangle$ *is defined by a set of context variables* $V = \{v_1, v_2, ..., v_n\}$. *Each variable* v_i *varies over a domain* $D_i = \{d_{i1}, d_{i2}, ..., d_{im_i}\}$ *with size* m_i.

Many variables either cannot be directly observed, or can only be partially sensed. If the heating mechanism is broken, we can sense that the heater is turned on, but we cannot observe if it actually started to heat the room, unless we have a reliable temperature sensor. Fortunately, many variables influence each other. For example, it is impossible to have a light turned on, if there is no electricity in the house; a location of a person and of the tool that she works with must be the same, etc. If these correlations are taken into account, even a few observed variables may give an overall, yet possibly incomplete, knowledge about the environment.

Definition 2 (Context, Interpretation). *For a given environment* $\langle V, D\rangle$, *a* context c *is a valuation of all variables in* V *with a non-empty subset* D^c *of* D. *If all variables* v_i *are assigned one and only one specific value in* D_i, *a context is called an* interpretation.

Non-emptiness ensures that a context is always possible in practice, i.e. each variable has at least one possible value.

We represent a context by enumerating its possible context variables values: $D^c_0, \ldots, D^c_n$, or, alternatively, as $v_0 \in \{d_{0l}, \ldots, d_{0k}\}$. We write $c.v_i$ to refer to i-th variable of c.

Our knowledge about an environment is described by a set of contexts $c_0, \ldots, c_n$. If for any two interpretations x, y s.t. $\forall c_i : x \in c_i \wedge y \in c_i$, it follows that $x = y$, then we have *complete* and *unambiguous* knowledge about the given environment. More than one interpretation represents an ambiguity or incomplete knowledge. Intuitively, each new sensor reading adds knowledge about the environment, thus it reduces the number of possible interpretations. Faulty contexts can be detected when an impossible situation is created, i.e. when there is no interpretation x, s.t. $\forall c_i : x \in c_i$.

For example, in Table Ia a portion of a smart home is modelled by 4 variables. In Table Ib, few pre-processing rules are defined that represent the inter-relation between the context variables. Note though, that it is not important how these rules are defined, as far as they result in a context information shown in Table Ic. Using these rules, from a reading that the light is on, we infer that the electricity is on, and if the TV is on as well, the channel is definitely not 'shows.'

A set of contexts $C = \{c_k\}$ is *consistent* if there exist at least one interpretation $x : x.v_i = d_{ij_i}, \forall i \in 1..n$ such that $d_{ij_i} \in c_k.v_i,\ \forall c_k \in C,\ \forall i \in 1..n$. A set of contexts is *inconsistent* otherwise.

Additionally, we define two relations over contexts:

- *Inclusion*: $c_1 \subset c_2$ iff $\forall i \in 1..n : \ c_1.v_i \subset c_2.v_i$ Inclusion can be viewed as a relation of a more precise and less precise contexts. If $c_1 \subset c_2$ then context c_1 is more precise, than c_2, in other words, each variable of c_1 contains less values that are possible.
- *Intersection*: $c_u = \bigcap_{j=1}^{k} c_j = c_1 \cap c_2 ... \cap c_k$ iff $\forall i \in 1..n : \ c_u.v_i = c_1.v_i \cap c_2.v_i ... \cap c_k.v_i$ An intersection of inconsistent contexts always equals to $\varnothing$. An intersection of consistent contexts is a context, that is at least as precise, that any of the originals: $\forall j \in 1..k \ \ c_u \subseteq c_j$.

To compactly represent all possible interpretations for a given set of contexts, we use relations defined in the previous section, thus forming a diagram with arrows representing inclusion relation. Any two contexts c_i, c_j are connected in the diagram, if $c_i \subset c_j$, and there is no such c_k, so that $c_i \subset c_k \subset c_j$.

The idea of putting contexts into the diagram structure is essentially an introduction of a compact representation of all possible interpretations of the environment. The "full domain" context is always at the top, meaning "no information is received; any situation is possible." Starting from the top and going down, contexts become more and more restrictive, with the most restrictive (as well as the most knowledgeable) contexts at the bottom. Formally, CCD is defined as follows:

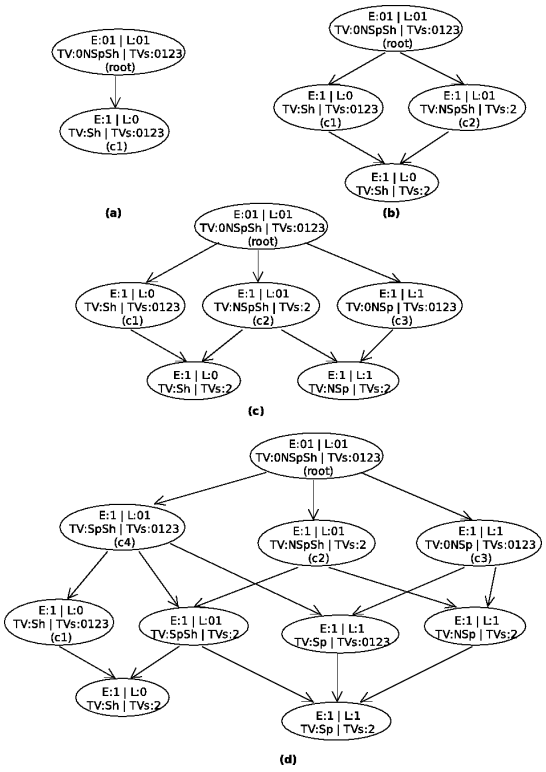


Figure 2: Example of context consistency diagrams.

Definition 3 (Context consistency diagram (CCD)). *Given an environment* $\langle V, D\rangle$ *and a set of contexts* $C_0 = \{c_k\},\ k \in 1..N$, *a* context consistency diagram (CCD) *is a tuple* $G = \langle C, E, r\rangle$, *where:*

- $r = D$, *is a special context, the* root;
- $C = C_0 \cup C_u \cup r$ *where* C_u *is the full set of intersections of a power set of* C_0.
- $E \subseteq C \times C$, *such that* $(c_2, c_1) \in E$ *iff* $\exists c_1, c_2 \in C : c_1 \subset c_2$ *and* $\nexists c_m \in C : c_1 \subset c_m \subset c_2$.

Contexts from a set C are vertices of the diagram and E is a set of directed edges. In a relationship $(c_1, c_2) \in E$, c_1 is called a **parent**, and c_2 is called a **child**. c_p is called a **predecessor** of c_c, and, respectively, c_c is called a **descendant** of c_p if either of the following holds:

1) $(c_p, c_c) \in E$
2) $\exists\{c_i\} \in C, i \in 1..k\ s.t.\ (c_p, c_1) \in E \wedge (c_k, c_c) \in E \wedge (c_i, c_{i+1}) \in E, \forall i \in 1..k-1$

We write $\psi(c)$ to denote the full set of descendants of c and $\Psi(c)$ to denote the full set of predecessors of c. Several important characteristics of the CCD directly follow from its definition:

1) An intersection of two consistent contexts $c_1 \in C$ and $c_2 \in C$ is a descendant of both contexts. $\exists c_u \in C,\ c_u = c_1 \cap c_2\ s.t.\ c_u = \psi(c_1),\ c_u = \psi(c_2)$.
   If $c_1 \in C$ and $c_2 \in C$ are inconsistent, then they do not have common descendants. $\nexists c \in C\ s.t.\ c = \psi(c_1),\ c = \psi(c_2)$.
2) If a set of contexts is empty, then CCD has only one root context. $C_0 = \varnothing \Rightarrow G = \langle r; \varnothing; r\rangle$
3) There is no context that is a predecessor of the root. A root is a predecessor of all other CCD contexts. $\forall c \in C : \nexists(c, r) \in E\ ,\ r \in \Psi(c)$

For a set $C_0 = \{c_1, c_2, c_3\}$, the corresponding set of intersections of its power set is equal to $C_u = \{c_1 \cap c_2, c_1 \cap c_3, c_2 \cap c_3, c_1 \cap c_2 \cap c_3\}$. For a set of contexts listed in Table Ic the corresponding CCD is shown on Figure 2.

V. Calculation of probabilities

When the CCD results in more than one interpretation, it is important to assess the likelihood of each interpretation. For a query (Figure 1), we provide answers for the following three possible requests: (i) the probability that a particular situation is true; (ii) the probability that a variable has a certain value; (iii) the dependence of variables on one another. In other words, the conditional probability that a certain variable has a certain value in case another variable has *a priori* known value.

We now describe how the CCD is used to address all above queries at any given moment of time. To calculate the probabilities mentioned above, we first need to introduce the concept of initial weight function $w_0(c)$. The initial weight function shows the importance of each original context $c \in C_0$. The weights depend on many things, among which are the infrastructure of sensor network; the importance of each sensor (the more important is the sensor, the more important is the context, associated with this sensor reading); and the number of times a particular context has been read. If the initial probabilities are unknown, we assume a uniform distribution, that is, any sensed information is equally likely. In a presence of additional information, other strategies for assigning weights may be chosen. The strategy should be chosen at the initial setup. The example in Figure 3 assigns the weight 1 uniformly to all contexts.

For all contexts that are not in C_0 the initial weight function equals to 0: $\forall c \notin C_0 : w_0(c) = 0$. The full weight function $w(c)$ for each context in a CCD is defined as $w(c) = w_0(c) + \sum_{\forall c_p:\ (c_p, c) \in E} w(c_p)$.

The full weight function takes into account that contexts that are consistent with each other should weight more than inconsistent ones. The idea is that consistent contexts form a consistent view on the situation, thus they all can be correct. While in the set of inconsistent contexts some are certainly faulty. So the full weight function rewards contexts for being consistent with others by increasing the weight of their descendants. The full weight of the CCD is the sum of weights of all its contexts: $w(G) = \sum_{c \in C} w(c)$

To calculate the probability that a variable has a certain value, we adapt the weight of the context to calculate the weight of each value of the variable inside a context. The context with several values of some variable assumes that

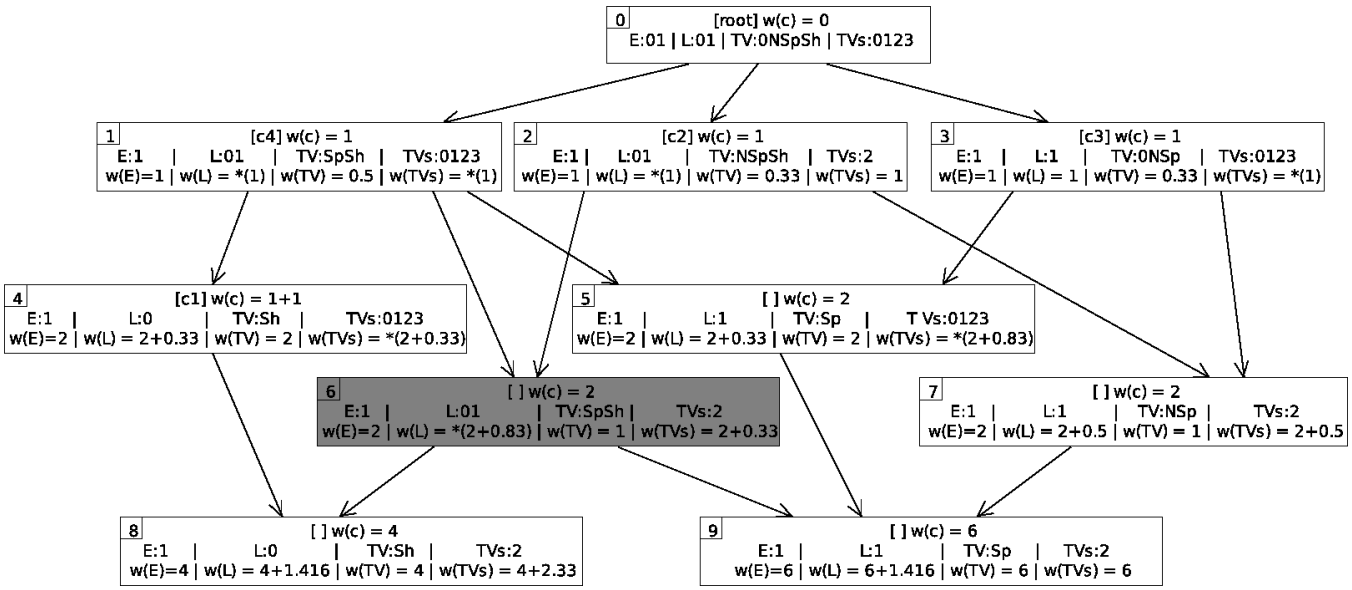


Figure 3: Assigning weights to the CCD in Figure 2d.

each of these values is equally probable, so we divide the weight of the context among all values for each variable: $w(c.v_i = d_{ij}) = w(c)/|c.v_i|,\ \forall d_{ij} \in c.v_i$. However, we do it only if the context actually knows something about the variable v_i. For example, if we got a sensor reading that the light is on, it tells nothing about the TV sound, so we do not split the context weight among TV sound values. If later we receive a context that tells us both that the light is on and TV sound is 2, then the first context *supports* the second one (since they are consistent), so we transfer the weight of the TV sound of a first context to a second one. For this we introduce the $\star$ value for a variable weight. This value means that the weight is transferred to the children of the context. Taking this into account, the weight of each value of each variable in a context is calculated by:

$$w(c.v_i = d_{ij}) = \begin{cases} \star & \text{if } c.v_i = D_i \\ \frac{w(c)+t(c.v_i)}{|c.v_i|} & \text{otherwise} \end{cases} \quad (1)$$

where $t(c.v_i)$ is a *transfer* (or *carrying*) value from the parents of the context:

$$t(c.v_i) = \sum_{\forall c_p:\ (c_p,c)\in E\ \&\ w(c_p.v_i)=\star} \frac{w(c_p) + t(c_p.v_i)}{|\{\forall c_c :\ (c_p, c_c) \in E\}|} \quad (2)$$

$\star$ also contributes towards the efficiency during CCD updates (Section VI). To calculate the final probabilities, we treat $\star$ differently depending on the context having children or not. If the context has children, the weight of a variable is fully transferred to them. Otherwise, we have no knowledge about the value of the corresponding variable, so each domain value gets an equal amount of the full weight, i.e. $\forall d_{ij} \in D_i$:

$$(w(c.v_i) = \star) \Leftrightarrow \begin{cases} w(c.v_i = d_{ij}) = 0 \\ \quad \text{if } \exists (c, c_c) \in E \\ w(c.v_i = d_{ij}) = \frac{w(c)+t(c.v_i)}{|c.v_i|} \\ \quad \text{if } \nexists (c, c_c) \in E \end{cases} \quad (3)$$

Now we can calculate the probability for each variable that it has a certain value:

$$pr(v_i = d_{ij}) = \frac{\sum_{\forall c \in C} w(c.v_i = d_{ij})}{w(G)} \quad (4)$$

As an example, we describe the calculation of weights on context 6 (grayed out) in Figure 3. The full weight of the context is 2, because $w(6) = w(1)+w(2)$. This weight fully goes to the sole value of the variable E, so $w(E = 1) = 2$. But for values of the variable TV, this weight is split equally in two, so each value $TV = Sp$ and $TV = Sh$ gets half of the full weight, or 1. The weight of the variable TVs is equal to 2.33, because it combines the weight of the context 6, and a third part of a transfer value from the context 1. The weight of the variable L is equal to $\star$,because this variable allows any value of the corresponding domain. So, this context does not assign any weight to values of L, but instead transfers it to its two children.

If we want to calculate the conditional probability that a certain variable has a certain value in case another variable has a particular value $pr(v_i = d_{ij}/v_c = d_c)$, we need to reduce weights in a CCD in such a way that only contexts that are compatible with $v_c = d_c$ have weights higher than 0. Also, for contexts, that allow other values for v_c, we need to correspondingly reduce their weight.

The conditional weight of each context is equal to

$$w'(c) = \begin{cases} \frac{w(c)}{|c.v_c|} & \text{if } d_c \in c.v_c \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

Similarly, the conditional weight of each variable is

$$w'(c.v_i) = \begin{cases} \frac{w(c.v_i)}{|c.v_c|} & \text{if } d_c \in c.v_c \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

Finally, conditional probability is equal to

$$pr(v_i = d_{ij}/v_c = d_c) = \frac{\sum_{\forall c \in C} w'(c.v_i = d_{ij})}{w'(G)} \quad (7)$$

where $w'(G) = \sum_{c \in C} w'(c)$

VI. Maintaining CCD

A. Algorithms

Next, we describe the algorithms for maintaining the CCD when new sensor data arrives. We start by introducing a few properties of the CCD that make the maintenance possible.

Property 1. *For a given set of contexts there is one and only one non-isomorphic representation of its CCD.*

It follows directly from the rules of construction. C_u is only dependent from C_0, and the root is always the same for the same variables and their domains. So $C = C_0 \cup C_u \cup r$ is always the same for a given C_0. For each pair of contexts in the CCD $c_1, c_2 \in C$ we use Definition 3 to determine if they are connected, i.e. if $\exists(c_1, c_2) \in E$. So, for each C_0 there is only one way to construct a tuple $G = \langle C; E; r \rangle$.

The actual context information changes rapidly and the CCD should be updated in real-time to always conform to it. Sensor readings arrive independently, and the CCD must be reconstructed to accommodate new information. After some time, obsolete contexts should be removed from the CCD to eliminate obsolete situations. Obviously, the CCD should not be constructed from scratch with each change in a contexts set. Instead a newly arrived context should be added to the existing CCD (by only changing the affected nodes), and an obsolete context and its obsolete descendants must be removed without affecting other parts of the diagram.

From the fact that for the given set of contexts there is only one CCD follow two more important properties.

Property 2. *The order of contexts addition does not change the resulting CCD.*

According to this property we can handle contexts updates one by one, without taking into account the order of their arrival, which can vary for asynchronous updates.

Property 3. *Adding and then removing a context does not change the resulting CCD.*

Properties 2 and 3 follow from the fact that the set C_0 is not ordered.

Algorithm 1 Adding context to CCD

```
1:  function AddContext(context, parent, weight)
2:  for all child ∈ parent.children do
3:      if child = context then
4:          W_0(child) ← W_0(child) + weight
5:          return
6:      else if context ⊂ child then
7:          AddContext(context, child, weight)
8:          return
9:      else if child ⊂ context then
10:         Remove link from parent to child
11:         Insert link from context to child
12:     end if
13: end for
14: Add link from parent to context
15: for all child ∈ parent.children\context do
16:     if isConsistent(context, child) then
17:         x ← context ∩ child
18:         AddContext(x, child, 0)
19:         AddContext(x, context, 0)
20:     end if
21: end for
```

Algorithm 1 contains the pseudocode of the addition of a new context to the diagram. It is started by running $AddContext(context, root, weight)$ (trying to add a new context directly under the root) and recursively descends to check all contexts that are consistent with a new one. The function $AddContext(context, parent, weight)$ is called only when a $context$ should be a descendant of a $parent$. Firstly it checks if a $context$ is already present as a child of a $parent$ (lines 3-5), if it is a child of a child (lines 6-8), or if some existing children of a parent should become children of a new context (lines 9-11). If the $context$ is not yet present and not a child of a child, then it is added as a new child, and all existing children are checked for consistency with it. If they are consistent, their intersection is created and recursively added to both contexts (lines 15-21). Note that an intersection context receives no initial weight.

Algorithm 2 shows the removal of an outdated context from the diagram. CCD has to be changed only if there are no other similar contexts and if it has only one parent (line 3), otherwise it must stay in the diagram as an intersection of its parents. The reduction of the CCD starts with removing a link from a $parent$ to a $context$ (line 5) and from a $context$ to all its children (line 7). We want all children of a removed $context$ to be added to its $parent$ directly. But we do not want to add a link from a $parent$ to a $child$, if they are already linked through different path. So on lines 8-10 we check if this is the case, and if it is not, we add a link (line 9). To be sure that no child without initial weight is left with a single parent, we recursively check all

Algorithm 2 Removing context from CCD

```
1: function RemoveContext(context, weight)
2: W_0(context) = W_0(context) − weight
3: if W_0(context) = 0 and |context.parents| = 1 then
4:    parent ← context.parents
5:    Remove link from parent to context
6:    for all ch ∈ context.children do
7:       Remove link from context to ch
8:       if ∄brth ∈ ch.parents s.t. brth ⊂ parent then
9:          Add link from parent to ch
10:      end if
11:      RemoveContext(ch, 0)
12:   end for
13: end if
```

children of a context for deletion (line 11).

Algorithm 3 calculates the probabilities in the CCD as described in Section V. The important part is line 7. $conditions$ is a context that contains a conditional situation, if we want to calculate a conditional probability. For finding unconditional probabilities, we put to $conditions$ the full domain context, similar to $root$. When we want to obtain probabilities in case that a certain variable has a certain value $v_c = d_c$, we produce a context $conditions$ in such a way, that we only allow one value for v_c, but all values for other variables: $conditions = \{v_c = d_c; \forall v_i \in v \backslash v_c : v_i = D_i\}$. This can be combined, to create a more sophisticated conditions. $coeff$ contains a percent of a context c that is contained in $conditions$. Later a weight of c and all its variables is reduced to this percent.

Lines 11-15 check the applicability of the $\star$ value.

Lines 18-25 go through all children of a $context$ and increase their weight. In case all parents are calculated, they are also added to the queue. Each context of the CCD will be queued exactly once, but only in a case it satisfies $conditions$ (otherwise it's conditional weight is 0).

B. CCD complexity

Explicit description of different interpretations in a CCD can potentially grow in space exponentially with the size of the environment. inconsistencies. However, there are several considerations that help to keep the size of a CCD reasonable.

The biggest growth of a CCD results from faulty contexts. While correct contexts tend to have the same descendants, faulty contexts will generate many new CCD nodes. With a growth of a CCD, one may discard contexts that support the most unlikely interpretations, as most probably they represent faulty or imprecise sensors.

Each environment in a CCD should only contain interdependent variables (i.e. associated by dependency rules, as in Table Ib). We split independent variables on non-intersecting subgroups and produce a smaller CCD for each subgroup.

Algorithm 3 Retrieve probabilities

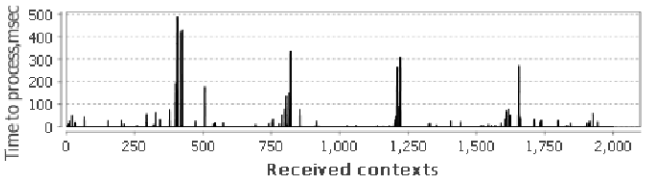
```
1: function RetrieveProbabilities(conditions)
2: Set all W_G, W(c.v_i), W(d_ij), t(c.v_i) to 0
3: Set all W(c) to W_0(c)
4: queue ← root
5: while queue is not empty do
6:    c ← queue.poll()
7:    coeff ← CalcCoefficient(conditions, c)
8:    W_G ← W_G + coeff * W(c)
9:    for all c.v_i do
10:      W(c.v_i) ← coeff * W(c) + t(c.v_i)
11:      if c.v_i = D_i and |cch ← c.children| > 0 then
12:         t(ch.v_i) ← t(ch.v_i) + W(c.v_i)/|cch| ∀ch ∈ cch
13:      else
14:         W(d_ij) ← W(d_ij) + W(c.v_i)/|c.v_i| ∀d_ij ∈ c.v_i
15:      end if
16:   end for
17:   Mark c as calculated
18:   for all child ∈ c.children do
19:      if child satisfies conditions then
20:         W(child) ← W(child) + coeff * W(c)
21:         if all child.parents are calculated then
22:            queue.add(child)
23:         end if
24:      end if
25:   end for
26: end while
27: return prob(d_ij) ← W(d_ij)/W_G for all d_ij
```

VII. Performance evaluation

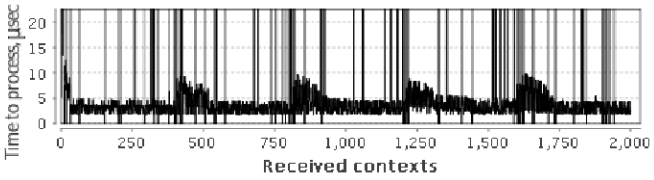
The experiments are performed on a Intel Core2Duo P7370 2GHz PC with 3 GB RAM running OS Ubuntu 10.04. The software is written in Java JDK 1.6. The test environment consists of a test generation part that generates situation and contexts, and a middleware part that collects contexts, maintains CCD, and calculates probability.

The experiments model a situation with 10 dependent sensors that sense the same type of data and are located in line one by one. The dependence is represented by a fact that the data of two neighboring sensors cannot differ for more than one measurement unit. Related real world situations include temperature or noise sensors. The context arrival rate is set to 0.05 seconds; lifetime is set to 6 seconds. The test generation part creates a situation. Contexts are generated based on it with a 5% error rate and are sent to the middleware part. Each 20 seconds the situation changes to capture the behavior of the CCD in a changing environment.

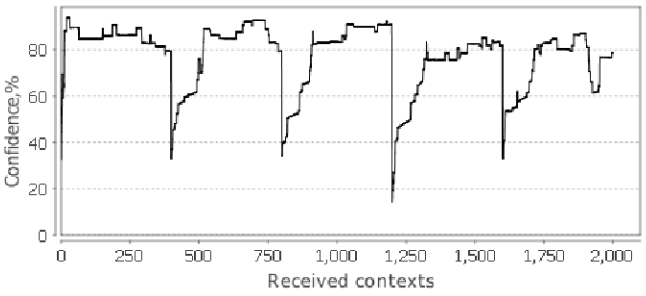
Figure 4 shows the adding of a new context to the CCD. The spikes in Figure 4a show that some contexts take over hundred times more time to process than average. Those are erroneous contexts, and contexts that are received after a change of a situation. Erroneous contexts require more



(a) Time to update the CCD with a new context



(b) Close view to the data in 4a. Correct context require little processing time.



(c) Confidence of the correct contexts with time

Figure 4: Performance experiment

time to calculate, as most of them imply that new nodes are added to the CCD. This also happens with a change of situation when new CCD nodes should be constructed. After receiving several new contexts the CCD "remembers" the new situation, and the time of context processing is reduced to the previous level. Figure 4b is a zoom of the bottom of the Figure 4a in close proximity. Most of the correct contexts already fit into the existing CCD structure, thus they require little processing time. Figure 4c shows the confidence in the correct context: the calculated probability of a correct situation. Drops in confidence when the situation changes represent that the CCD mainly remembers the obsolete situation. With the arrival of new contexts the new situation quickly becomes the most probable one.

CCD are maintained continuously during the work of context-aware applications. Thus it is important that CCD maintenance computation efforts remain on the same level and do not increase with system up-time. Figures show that this is indeed the case. Once the middleware reaches its normal workload, the time of context processing remain on the same level over time, as can be seen in Figure 4b.

VIII. Concluding remarks

We introduced Context Consistency Diagrams, a novel data structure for reasoning about situations with incomplete knowledge of the environment and context conflicts that cannot be unambiguously resolved. The CCD allows for the computation of the probability of a certain interpretation, the value of a particular variable (or set of variables), and conditional probability over different dependent variables. Our experiments show that the CCD can be efficiently maintained and computed in real-time.

Acknowledgment

The research is supported by the EU projects Smart Homes for All, contract FP7-224332 and Greener Buildings, contract FP7-258888.

References

[1] E. Kaldeli, E. Warriach, J. Bresser, A. Lazovik, and M. Aiello, "Interoperation, Composition and Simulation of Services at Home," in *ICSOC*, vol. 6470, 2010, pp. 167–181.

[2] S. R. Jeffery, M. Garofalakis, and M. J. Franklin, "Adaptive cleaning for RFID data streams," in *Proc. of the Int. Conf. on Very Large Data Bases*, 2006, pp. 163–174.

[3] C. Xu, S. C. Cheung, W. K. Chan, and C. Ye, "Partial constraint checking for context consistency in pervasive computing," *ACM Trans. Softw. Eng. Methodol.*, pp. 1–61, 2010.

[4] Y. Huang, X. Ma, X. Tao, J. Cao, and J. Lu, "A probabilistic approach to consistency checking for pervasive context," in *EUC '08: Proc. IEEE/IFIP Int. Conf. on Embedded and Ubiquitous Computing*, 2008, pp. 387–393.

[5] Y. Bu, T. Gu, X. Tao, J. Li, S. Chen, and J. Lu, "Managing quality of context in pervasive computing," in *QSIC '06: Proc. Int. Conf. on Quality Software*, 2006, pp. 193–200.

[6] Y. Bu, S. Chen, J. Li, X. Tao, and J. Lu, "Context consistency management using ontology based model," ser. Lecture Notes in Computer Science, 2006, vol. 4254, pp. 741–755.

[7] C. Xu, S. C. Cheung, W. K. Chan, and C. Ye, "Heuristics-based strategies for resolving context inconsistencies in pervasive computing applications," in *Proc. of the 28th Int. Conf. on Distributed Computing Systems*, 2008, pp. 713–721.

[8] K. Henricksen and J. Indulska, "Modelling and using imperfect context information," in *Proc. of the 2nd IEEE Annual Conf. on Perv. Computing and Communications*, 2004, p. 33.

[9] H. Lu, W. Chan, and T. Tse, "Testing pervasive software in the presence of context inconsistency resolution services," in *Proc. Int. Conf. on Software engineering*, 2008, pp. 61–70.

[10] Y. Huang, X. Ma, J. Cao, X. Tao, and J. Lu, "Concurrent event detection for asynchronous consistency checking of pervasive context," in *IEEE Int. Conf. Pervasive Computing and Communications*, 2009, pp. 1–9.

[11] H. Kong, G. Xue, X. He, and S. Yao, "A proposal to handle inconsistent ontology with fuzzy owl," in *Proc. WRI World Congress on CS and Inf. Eng.*, vol. 1, 2009, pp. 599–603.

[12] F. Marcelloni and M. Aksit, "Leaving inconsistency using fuzzy logic," *Information and Software Technology*, 2001.