SWAN-Song: a Flexible Context Expression Language for Smartphones

Nicholas Palmer, Roelof Kemp, Thilo Kielmann, Henri Bal
VU University Amsterdam
De Boelelaan 1081
1081HV, Amsterdam, The Netherlands
{palmer,rkemp,kielmann,bal}@cs.vu.nl

ABSTRACT

The rise of smartphones, with numerous on board sensors, significant processing power, and multiple networking technologies, has finally created an environment perfect for contextual applications.

While examples of such applications exist today, it has been recognized that an intermediate layer between applications and sensors based on contextual expressions *simplifies* the creation of new context aware applications and sensors as well as *streamlines* evaluation of such expressions.

Key to such an intermediate context layer is a domain specific language in which the individual context conditions are combined into an expression that is meaningful to the contextual application.

To this end we present SWAN-Song, a domain specific language for context expressions. The addition of *history windowing* and *history reduction* in expressions, not found in similar languages such as CMQ or AnonyTL, significantly improves the expressivity of SWAN-Song, and also enables smart evaluation of such expressions through delaying re-evaluation and/or temporarily turning off sensors.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*Domain Engineering*

Keywords

Context, Domain Specific Language, Energy Efficiency, Mobile, Smartphone, Sensing

1. INTRODUCTION

For two decades researchers have held out context awareness as uniquely positioned to make computational devices smarter and more ubiquitous. Weiser's well known vision[11] of devices that melt into the background relies on devices that understand the location and situation in which they find themselves, and can behave in intelligent ways based on this information. Much research has been conducted in the intervening 20 years on how to make use of context in applications, while hardware technologies have dramatically changed the landscape of what is possible today.

Foremost amongst the developments of the last two decades is the rise of the smartphone as a general purpose computing device. Arguably beginning with the Apple iPhone, but certainly continuing with a diversity of Android powered devices, the smartphone has exploded onto the computing landscape. The plethora of smartphone devices contain advanced processors, multiple networking technologies and advanced sensing capabilities that researchers of two decades ago only dreamed of[10].

Currently mobile platforms offer low level access to the on board sensors through various interfaces. Though these interfaces enable the creation of context aware applications, we agree with [8] that the current situation suffers from both *poor abstractions*; the focus of programmers should be to construct high level context expressions and *poor programming support*; writing a context aware application requires expertise in sensor data gathering, processing, and expression evaluation.

In our ongoing work on the SWAN (Sensing with Android Nodes) project we address both of these problems through an intermediate middleware layer that offers powerful context abstractions to developers, as well as rich programming support in the form of a sensor maker tool, an expression creator, and automatic offloading for networked sensors. Out of the box SWAN contains 20+ sensors and supports plug-ins from 3rd party sensors.

In this paper we focus on the domain specific language of the SWAN project: SWAN-Song. SWAN-Song provides an abstraction of context much like other domain specific context languages such as SeeMon's CMQ[5] and AnonySense's AnonyTL[3], where complex context expressions can be built using logic operators and comparators. SWAN-Song extends the expressivity of these languages by adding support for *configuration*, *math operators*, and *history windowing*, i.e. look at a series of values in a history window rather than only the current value, similar to the WITHIN and FOR operators found in CITA[8]. In doing that we go beyond what is available in CITA because SWAN-Song allows such a series of readings to be used in a comparison where various history reduction modes, such as MIN, MAX, etc., are used to specify how the series should be compared.

Because values remain in a history window for an extended period of time, an expression evaluation engine that implements monitoring of SWAN-Song expressions can employ smart evaluation strategies to reduce both the processing for evaluation as well as the cost of sensing itself.

Although we developed SWAN-Song for the SWAN project, the language and evaluation strategies are applicable beyond our own system. The main contributions of this paper are:

- SWAN-Song language: we present a context expression language that supports logic combinations and comparator op-

PhoneSense'12, November 6, 2012, Toronto, ON, Canada.

erations, and extend it with math combinations, history windowing and history reduction operations as well as sensor configuration.

- SWAN-Song evaluation strategies: we present two strategies for smart evaluation of SWAN-Song expressions; *defer until* and *sleep and be ready*.

The remainder of the paper is organized as follows. Section 2 discusses the related work. We then present the SWAN-Song language in Section 3 and the evaluation strategies in Section 4. We conclude and outline our future work in Section 5.

2. RELATED WORK

Various middleware for context aware computing have been proposed, ranging from semantic web technologies [9, 6], to social oriented systems[4] and people-centric sensing middleware [3, 1]. While the interface of some of these systems is SQL like, such as Kobe[2], there are several projects that come with a domain specific language, among which SeeMon's[5] CMQ and AnonySense's[3] AnonyTL and some language constructs in MIT's CITA[8]. We selected to compare SWAN-Song throughout the paper against the most relevant other language: CMQ.

The SeeMon system can efficiently evaluate CMQ expressions, through computing essential sensor sets and creating an index, which can be used for lookups when new sensor readings come in, so that only a subset of the expressions have to be re-evaluated. Likewise, with the same goal of efficient evaluating in mind, but with different smart evaluation techniques, SWAN-Song can reduce the cost for evaluation in an evaluation engine and even allow sensors to be turned off for awhile.

3. SWAN-SONG

In this section we present the grammar and meaning of the SWAN-Song language. The language is a form of zeroth-order logic, which allows the description of a particular combination of contextual information using sensor based expression predicates, which can be combined using math, comparison, and logic operators, and a specification of a history window and history reduction operator.

3.1 Sensor and Constant Predicates

At the core of SWAN-Song there are the *sensor predicates*, which give access to the values of various sensors. Whereas sensor predicates in CMQ only specify the type of the sensor (e.g. temperature, location), we note that on mobile platforms a single sensor can often produce multiple context types. For instance, a GPS sensor produces not only longitude and latitude, but also speed and altitude. Another difference from CMQ – that was designed around non-configurable external sensors – is that on mobile platforms it is often possible to *configure* sensor properties, such as the sample interval. Because of this, in addition to the CMQ sensor name, we require SWAN-Song context predicates to support both a path within a sensor to precisely indicate the context type, and an optional configuration part that specifies how the sensor should be used.

The third and most important difference between SWAN-Song and CMQ predicates is that SWAN-Song predicates explicitly allow for time windowing. In contrast to only considering the *current* value of a sensor, SWAN-Song predicates allow for the definition of a history window (e.g. the last 5 minutes). The result of such a predicate is by nature a series of data. Additionaly, SWAN-Song allows a sensor predicate to specify a history reduction operator that defines how the series of data are dealt with further on in the expression.

Formally, a sensor predicate is made up of two required components and three optional components. Required are a sensor entity and a value path. Optional components are a list of one or more configuration parameters, a history window and a history reduction mode.

An example of a sensor predicate is the following:

```
wifi:ssid?sample_interval=10m {ANY, 1h}
```

The first component (`wifi`) is the sensor entity that defines what sensor the predicate references. The second is the value path (`ssid`) which specifies the value within a given sensor. Next there is an optional list of configuration options for the sensor represented as a series of key value pairs separated by an `&`. These options can be used to specify sensor specific configuration information, which varies on a per sensor and per predicate basis (`?sample_interval=10m`).

A sensor predicate has an implicit time window on the values it considers when the expression is evaluated. The default windowing is over the last second, however expressions may specify a different window by appending a time in brackets along with the sensor predicate (`1h`).

Because of this windowing of sensor values, sensor predicates represent an array of time-stamped values. What the user wants to know is if the screen was on in the last 5 minutes, represented by the expression: `screen:on {5m} == true`. Since the screen may have been on and off in that interval it is unclear if this is true or false. To solve this issue, our system includes a history reduction mode (`ANY`). The default mode is `ANY`, which selects any value that makes the comparison true as the value of the sensor. This parameter is placed within the brackets after the expression, along with the history length if present. `ANY` and `ALL` are logically equivalent to CITA's `WITHIN` and `FOR`. With this system it is easy to ask questions like has the screen been constantly on for the last 5 minutes simply by changing the reduction strategy to `ALL` giving the expression: `screen:on {5m, ALL}`. Next to `ANY` and `ALL`, the system offers modes for value paths that are numeric in the form of `MIN, MAX, MEAN`, and `MEDIAN`, which will select or calculate the appropriate value. We have not seen an equivalent combination of features in other expression languages in the literature. In particular CMQ has no concept of how to reduce history in the time window, and CITA only offers our ANY and ALL modes. In addition to sensor predicates, SWAN-Song supports constant predicates of various types including numeric, string and boolean values, of use in more complex expressions.

3.2 Complex Expressions

Constant and sensor predicates can be combined into higher level expressions. The language includes common comparison operators including `==`, `>=`, `>`, `<=`, `<` and `!=`, and also string operators like `regex` for performing a regular expression match on strings, as well as `contains`, `startsWith`, and `endsWith`. To the best of our knowledge there is no such a rich vocabulary of comparators in the literature. Another powerful extension of functionality in the language is the support of basic math operators over numeric sensor values and constants. Similarly to what is available in other languages, SWAN-Song supports the logic operators binary AND and OR and unary negation allowing complex expressions to be constructed. An example that includes comparators, history reduction operators, math operators and logic operators is shown below and detects the battery running down quickly by a background service:

```
 screen:on {ALL, 1h} == false &&
(battery:level {MAX, 1h} -
 battery:level {MIN, 1h}) > 25
```

Like CMQ, SWAN-Song is a tri-state language, in that all expressions evaluate to `TRUE`, `FALSE` or `UNDEFINED`. This third state is

important, since an expression written for one device may reference sensors which are not available on another device, or the user may have forced a particular sensor to be off, for instance turning off Bluetooth to save battery. Reporting a true of false value for an expression involving such a sensor would be incorrect. Thus SWAN-Song is designed to handle the uncertainty inherent to operating on real devices. The UNDEFINED state allows SWAN-Song to avoid giving misleading results when the value of an expression is not known, perhaps due to missing sensor data.

Finally, we note that while we do not offer quantification operators as is found in first-order logic based expression languages[7], we are able to express even more powerful time based expressions due to the history window and history reduction operators, where as the prior first-order logic based expressions can only be used to reason about the current context.

4. SMART EVALUATION

Now that we have outlined the SWAN-Song language we discuss the implications of the language on a runtime system that evaluates SWAN-Song expressions. In the overall SWAN project we aim at building a system for mobile platforms with limited resources. Therefore it is vital that the expression evaluation engine is as efficient as possible; it should minimize evaluations of single expressions to scale to a large number of expressions and it should prevent resources being spent on redundant sensing. The properties of SWAN-Song allow for two important optimizations in the evaluation engine: reduction of evaluation by deferring re-evaluation and disabling sensors for a while, both of which we have implemented in the evaluation engine of SWAN.

4.1 Defer Until

Once an expression is registered with the evaluation engine, it is repeatedly being evaluated. Each evaluation will result in one of the values TRUE, FALSE, or UNDEFINED. Only when the evaluation result differs from the previous result the registrar of the expression is notified.

Because a new reading might change an expressions evaluation result, a new evaluation is typically scheduled upon any new sensor reading coming in. However, in some cases we know a priori that the new reading will not affect the result of the expression and therefore can delay a new evaluation.

For example, consider when we monitor the maximum value of a sensor over the last minute to see whether it is greater than a certain constant threshold and we find that just 5 seconds ago the sensor reported a value that is above our threshold. Then we know that any evaluation in the next 55 seconds of this expression will result in TRUE. Therefore we can safely defer the evaluation for 55 seconds. Note that although we can defer evaluation, we still need to store the sensor readings; the next evaluation after 55 seconds needs a full 60 seconds history window.

In the above example it is appropriate to defer re-evaluation, but in general finding out whether it is possible to defer the re-evaluation of an expression requires the inspection of these four variables:

- the history reduction mode
- the comparator
- the current result (TRUE, FALSE, or UNDEFINED)
- the operands of the comparator (constant or not)

For instance, the above combination of comparing the MAX of a context value with a constant threshold using the > operator while the result is TRUE allows for deferring evaluation. However, if we were looking at the history reduction mode MIN (while keeping >, TRUE), then any new reading can make the minimum less than our threshold and render the expression FALSE. In that case we cannot defer evaluation. Also if the operator is different in the case of MAX, <, TRUE we cannot defer evaluation, because any new reading can make the maximum greater than our threshold and render the expression FALSE.

Table 1: This table shows the history reduction modes for which we can calculate a *defer until* time depending on the comparator and the expression result. This table is valid for expressions with a context value on the left side and a constant value on the right side.

	comparator	result	hist. reduction mode
A1	>, >=	TRUE	MAX, ANY
A2		FALSE	MIN, ALL
A3	<, <=	TRUE	MIN, ANY
A4		FALSE	MAX, ALL
A5	==, match, contains,	TRUE	ANY
A6	startsWith, endsWith, regex	FALSE	ALL
A7	!=	TRUE	ALL
A8		FALSE	ANY

Table 2: Method for calculating the defer until time for logic expressions depending on the logic operator and the values of expressions A and B.

	A	B	&&	||
B1	T	T	MIN($defer_A$,$defer_B$)	MAX($defer_A$,$defer_B$)
B2	T	F	MIN($defer_A$,$defer_B$)	$defer_A$
B3	F	T	MIN($defer_A$,$defer_B$)	$defer_B$
B4	F	F	MAX($defer_A$,$defer_B$)	MIN($defer_A$,$defer_B$)

We made an exhaustive list of all combinations of history reduction modes, comparators and results and for each item we determined whether a new reading can change the evaluation result or not. We used the results of this analysis to create a look up table to determine whether deferring next evaluations is applicable. Table 1 shows the cases in which we can defer evaluation.

Once we know that we can defer evaluation, we need to determine for how long. We take the timestamp for the newest reading that given the reduction mode determines the result value and add the history window length to it. The resulting time is the new evaluation time. Note that for the MAX and MIN modes we do not necessarily consider the actual maximum or minimum, since a reading with a lower value than the maximum, but still above the threshold, and with a later timestamp leads to a longer defer until time.

In addition to using historical values to delay evaluation, we can also make use of the time sensor; the one special sensor that knows ahead what its values will be in the future. Once a (sub-)expression contains time we can immediately determine if and when the evaluation result is going to change and defer evaluation until then.

4.1.1 Sibling Deferring

Furthermore, when we have logic expressions built out of other expressions we can sometimes defer evaluation of one of the children even longer, because of the logic operator and the defer until of the other child. For instance, if we have expression A AND expression B resulting in the value FALSE, then we know that both A and B have to change into TRUE to make the logic expression change into TRUE. Although the defer until of one of the sub-expressions can be much shorter than the other one, we can safely delay evaluation of the

logic expression (and thus its sub-expressions) until the maximum defer time of both sub-expressions. Table 2 shows which method should be used in what scenario to determine the defer until of a logic expression.

Although the defer until time of some of the logic expression combinations is the minimum of both its children, it does not mean that once the logic expression gets evaluated both children have to be evaluated. All sub-expressions will cache their results and return this upon evaluation when its own evaluation can still be deferred.

Figure 1 gives an example of how, defer until can be used. Note that the sub-expression `heartrate:bpm{MEAN, 60s} > 180` on its own requires intensive processing for evaluation; every new reading causes the average to change. The evaluation of its sibling, the speed expression, can be deferred according to rule A2 from Table 1. Through sibling deferring (rule B2 from Table 2), the *evaluation* of the heartrate expression can be deferred for 5 minutes. The swimming expression in turn can be deferred for 2 hours (rule A5) and then through sibling deferring (rule B3) the running expression and child expressions can also be deferred 2 hours.

4.2 Sleep And Be Ready

We can go a step further in optimizing and not only minimize *evaluation* of sensor readings, but also *stop gathering* sensor data when possible.

Intuitively, one would argue that as long as we can defer evaluation of an expression we also do not need the sensor readings. However, if we turn on the corresponding sensors at the moment a new evaluation is needed, we need the complete history for this evaluation. Thus, we need to instruct sensors to wake up at a specific point in time, such that when the expression is re-evaluated, all the required history is available. We call this optimization *Sleep And Be Ready*.

To illustrate that the evaluation strategy matters for the Sleep And Be Ready optimization, we recall the example in Figure 1, but now focus only on the sub-expression *running* that results in `FALSE`. Because both of its children result in `FALSE`, we can apply logic short-cutting and terminate evaluation after evaluating one of the children. While any evaluation order of the children results in such an *evaluation* optimization, we need a specific evaluation order to reach a Sleep And Be Ready optimization.

If the heartrate expression is evaluated first, we will not evaluate the speed expression, because of short-cutting. Then the running expression can try to use the Sleep And Be Ready optimization and instruct the gps sensor to sleep and be ready at the next evaluation time. But because there is no defer until time for the heartrate expression, the speed expression needs to be ready immediately and the gps sensor cannot be turned off. In contrast, when the speed expression is evaluated first, we find that we can defer evaluation for 5 minutes. The running expression can thus inform the heartrate sensor to be ready after 5 minutes. Internally the heartrate expression considers its own history window and will sleep for 4 minutes and 30 seconds, so that after 5 minutes its history window of 30 seconds is filled.

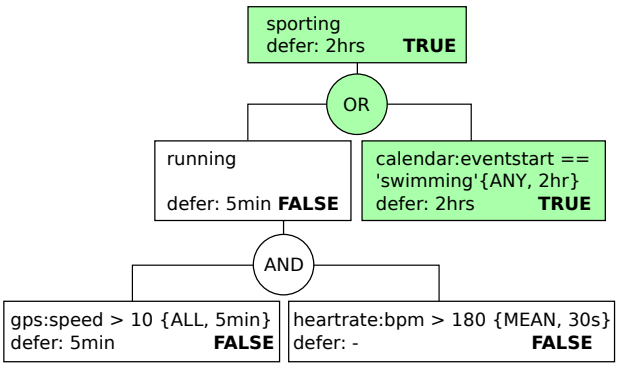

Figure 1: Expression tree including evaluation state and computed defer time. The expression defines *sporting* as either *swimming* or *running*. The swimming context is determined based on the calendar, the running context is derived from a combination of the heartrate sensor and the gps.

From this example we learn that it is best to adopt the following strategy to determine which child expression to evaluate first. For both nodes we calculate the sleep time, which is: $t_{sleep,me} = t_{deferuntil,other} - t_{history,me}$

We then pick the one with the lowest sleep time, and evaluate that one first. In our future work we will also take *warm up* times for sensors into account (e.g. the time it takes before a sensor produces its first reading, such as getting a GPS fix).

5. CONCLUSIONS AND FUTURE WORK

The rise of smartphones has finally created an environment ideal for contextual applications. The promise of these devices is that they will become even smarter, adapting to users situations in order to make the device integrate seamlessly with the users desires. In this paper we have described SWAN-Song, our domain specific language for expressing contextual conditions. This language is part of our SWAN platform for Android powered smartphones. Other than existing languages, SWAN-Song allows for history windowing and history reduction, supports math operators in the language and enables sensor expressions to contain a specific configuration. Inherent to history windowing and history reduction operators are two smart evaluation strategies that can be employed by an expression evaluation engine; the *defer until* strategy to minimize the number of evaluations per expression and the *sleep and be ready* strategy to turn off sensors used in sub-expressions.

Currently the research in the SWAN project is in an early stage. Although some simple applications on top of the system show promising results for scalability and power efficiency, in our future work we will create more context aware applications to provide an extensive evaluation of how much the smart evaluation strategies of SWAN-Song contribute to scalability and power efficiency. Our future work will also explore distributed context expressions by adding support for cross device expressions. This work will have to address communication failures inherent to distributed computing, for which we anticipate to use the `UNDEFINED` state. Finally, we are exploring the use of Statistical Machine Learning (SML) techniques over collected sensor data in order to automate the generation of SWAN-Song expressions for particular contextual situations.

In conclusion, we argue that SWAN-Song represents an important step forward in context expression languages, combining lessons learned from other prior work into a very expressive, unified language suitable for smart evaluation.

6. REFERENCES

[1] AHARONY, N., PAN, W., IP, C., KHAYAL, I., AND PENTLAND, A. Social fMRI: Investigating and shaping social mechanisms in the real world. *Pervasive and Mobile Computing* (2011).

[2] CHU, D., LANE, N., LAI, T., PANG, C., MENG, X., GUO, Q., LI, F., AND ZHAO, F. Balancing energy, latency and accuracy for mobile sensor data classification. In *Proc. of the*

9th ACM Conf. on Embedded Networked Sensor Systems (2011).

[3] CORNELIUS, C., KAPADIA, A., KOTZ, D., PEEBLES, D., SHIN, M., AND TRIANDOPOULOS, N. Anonysense: privacy-aware people-centric sensing. In *Proceeding of the 6th international conference on Mobile systems, applications, and services* (2008).

[4] GUPTA, A., KALRA, A., BOSTON, D., AND BORCEA, C. MobiSoC: a middleware for mobile social computing applications. *Mobile Networks and Applications 14* (2009).

[5] KANG, S., LEE, J., JANG, H., LEE, H., LEE, Y., PARK, S., PARK, T., AND SONG, J. Seemon: scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. In *Proceeding of the 6th international conference on Mobile systems, applications, and services* (2008).

[6] LEE, S., CHANG, J., AND LEE, S. Survey and trend analysis of context-aware systems. *Information-An International Interdisciplinary Journal 14* (2011).

[7] RANGANATHAN, A., AND CAMPBELL, R. An infrastructure for context-awareness based on first order logic. *Personal and Ubiquitous Computing* (2003).

[8] RAVINDRANATH, L. S., THIAGARAJAN, A., BALAKRISHNAN, H., AND MADDEN, S. Code In The Air: Simplifying Sensing and Coordination Tasks on Smartphones. In *HotMobile* (February 2012).

[9] RHO, T., APPELTAUER, M., LERCHE, S., CREMERS, A., AND HIRSCHFELD, R. A context management infrastructure with language integration support. In *Workshop on Context-oriented Programming* (2011).

[10] SCHMIDT, A., BEIGL, M., AND GELLERSEN, H. There is more to context than location. *Computers & Graphics 23* (1999).

[11] WEISER, M. The Computer for the Twenty-First Century. *Scientific American 265*, 3 (1991), 94–104.