A Domain-Specific Language for the Specification of Adaptable Context Inference

André C. Santos, Pedro C. Diniz
INESC–ID
Lisbon, Portugal
acoelhosantos@ist.utl.pt, pedro@esda.inesc-id.pt

João M. P. Cardoso
FEUP – University of Porto
Porto, Portugal
jmpc@acm.org

Diogo R. Ferreira
IST – Technical University of Lisbon
Lisbon, Portugal
diogo.ferreira@ist.utl.pt

***Abstract*—Context-aware mobile applications can benefit from context inference adaptation based on run-time operating conditions, such as battery life or sensor availability. Developing applications with such adaptable behavior, however, is notoriously cumbersome, as developers need to deal with low-level system interfacing and programming issues. In this paper we describe a domain-specific language (DSL) and a middleware infrastructure to support the specification, deployment and maintenance of run-time adaptable context inference processes. We illustrate the benefits of our approach via a case study, highlighting the new abstractions that facilitate the specification of adaptable behavior using different algorithms and the corresponding varying parameter settings, with a specific goal of minimizing the energy while maintaining acceptable end-application performance and accuracy.**

***Index Terms*—mobile devices; context-awareness; adaptable context inference; domain-specific language; middleware.**

I. Introduction

Mobile applications that make use of context information can provide a rich and more personalized set of services, such as tour guides [1], support for health care systems [2] or enhanced social networking [3]. To support these advanced services, mobile applications acquire context information, such as user location or user activity, through inference processes that rely on sensor data analysis and reasoning methods ranging from simple operations to sophisticated algorithms [4].

The increasing sophistication of these applications creates a tremendous pressure on the limited resources of mobile devices, in particular energy, making it very desirable to take into account the run-time operating conditions when performing inference. Mobile applications can leverage adaptations to keep context inference processes running despite changes in operating conditions (e.g., battery running low) or application requirements (e.g., increase context accuracy).

Possible adaptations include the use of different processing algorithms; different algorithm parameters tuned to specific contexts; distinct sensors that provide similar data; or simply different periods at which the inference is computed. As an example, Fig. 1 illustrates the impact of power consumption and CPU load on a mobile device (Nokia N95 smartphone) for a user activity context inference technique using a Fast-Fourier-Transform (FFT) and a k-Nearest-Neighbor (kNN) classifier over different windows of accelerometer data. Within this technique power consumption and CPU load decrease, but accuracy improves, as FFT processing window size increases.

As can be seen, there is a substantial impact on the operating conditions of the mobile device by changing a key algorithm parameter of the context inference process. Considering windows of 512 and 2048 samples, in the latter the accuracy increases slightly (82% to 89%) while almost requiring double the resources. In fact, in many situations, the inference process could opt for different algorithms and/or parameter settings that provide alternative inference configurations exhibiting reduced resource use without any significant loss in context inference accuracy and thus of application performance.

Unfortunately, the current development approaches for context-aware applications are still too rigid. At present, if developers want to implement such adaptations, they must do it by engaging in a complex, time-consuming and thus error-prone programming efforts. To mitigate these issues, we propose an approach for the development of context-aware applications that provides high-level abstractions for specifying the run-time adaptation of the context inference processes. Specifically, this paper makes the following contributions: (1) a platform-independent domain-specific language (DSL) for the specification of dynamically adaptable context inference processes; (2) a middleware platform for infrastructural support responsible for the interpretation and execution of context inference processes specified in the DSL; (3) the evaluation of the proposed approach in application scenarios and in a case study application. We expect that the joint approach of a DSL and a middleware will lead to an easier development of adaptable context inference, allowing mobile context-aware applications to deliver acceptable performance and enhanced functionality while optimizing resource usage and application resiliency despite changes in run-time operating conditions.

The remainder of this paper is organized as follows. Section II presents an overview of the middleware and DSL approach, which are further detailed in Sections III and IV, respectively. Section V presents a case study. We survey related work in Section VI, and conclude the paper in Section VII.

II. Approach Overview

The architecture of our approach, depicted in Fig. 2, is based on a specification DSL and a supporting middleware infrastructure, targeted for mobile device environments. The approach aims at enabling the use of adaptable behavior in

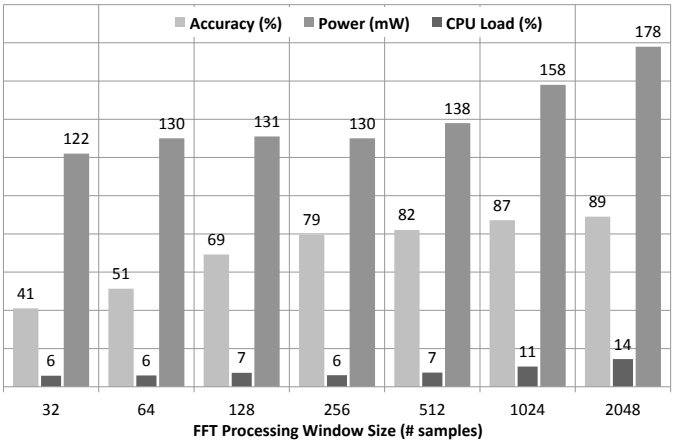

Fig. 1. Average power consumption and CPU load for different FFT processing window sizes. Measurements acquired using the Nokia Energy Profiler (NEP) on a Nokia N95 smartphone.

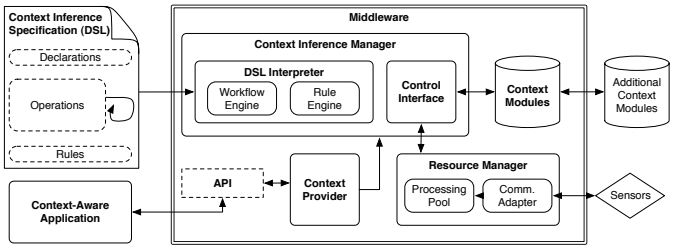

Fig. 2. Architecture of the proposed DSL and middleware approach.

the context inference process, through strategies and rules that dynamically control the inference workflow.

The DSL exposes the concepts of domain components, allowing the inference process to be concentrated on a set of clearly exposed inference operations and adaptation rules. Furthermore, support for the DSL is provided by a middleware infrastructure that interprets, executes and implements the various DSL elements in run-time. The DSL component is of paramount important as it is platform-independent promoting reusability, robustness, and enhanced productivity across different platforms. As it captures the semantics of a domain, the DSL provides key abstractions to specify adaptable inference, such as: defining input and output inference parameters; default implementation execution; and rules for adaptation logic of the inference workflow. An example of a DSL description associated with the case presented in Fig. 1 is shown in Fig. 3. The DSL defines how and when the inference process is computed (Fig. 3, lines 3–6) and the rules defining strategies to adapt the inference when specific events or conditions occur, i.e., when the battery of the device reaches a low level, when CPU reaches high load or when the inference elapsed time takes longer than the defined period (Fig. 3, lines 8–15). Adaptations consider computing the inference technique with a lower value for the number of samples parameter (Fig. 3, line 10), or increasing the inference period (Fig. 3, line 13).

Using our approach, the use of context information by applications becomes transparent, as adaptable inference behavior

```
 1: infAct IS fftKnnInf(FftNrSamples=2048){IDLE,WALK,JOG};
 2:
 3: RUN[period=1sec]{
 4:    // default implementation
 5:    activityContext = infAct();
 6: }
 7:
 8: RULES{
 9:    EVENT: ENERGY.LEVEL.LOW || CPU.LOAD.HIGH{ // Rule A
10:       infAct IS fftKnnInf(FftNrSamples=512);
11:    }
12:    EVERY RUN: RUN.ELAPSEDTIME > RUN.period{ // Rule B
13:       RUN.period = 2sec;
14:    }
15: }
```

Fig. 3. DSL code for adaptable user activity context inference.

is externally defined in the DSL using a specific syntax with domain semantics. As the DSL specification is implemented by the middleware, the application developer can thus focus on the main application logic since the context-related concerns are encapsulated by the middleware components, and require the middleware to deliver, by periodic subscription or by individual queries, the generated context information. The use of this approach implies specifying the inference process and adaptations using the DSL; providing the middleware with the required methods for inference; and also registering for context notifications or simply querying for contexts.

III. Middleware Infrastructure

The middleware provides applications and the DSL with a supporting infrastructure for the development and use of adaptable context inference (see Fig. 2), supporting abstractions that transparently provide services to deal with issues from low-level details to commonly used operations. The middleware is thus dependent on the target platform and execution environment. Internally, the middleware is composed of several components, namely a *context inference manager*, a *DSL interpreter*, a *context provider*, a *resource manager* and a number of *context modules*. Collectively the components coordinate the inference process, implement the DSL specification, manage resources, and provide context to applications.

In order to obtain context information, a context-aware application needs a mechanism for interfacing with the middleware platform. The middleware provides a set of APIs to support the interaction between the context provider and the application. The context provider is the interface middleware component responsible for supplying context information to applications by on-demand context queries or event subscriptions involving periodic updates. The context inference manager is the middleware component that controls the inference process through the DSL interpreter and the control interface sub-components. The DSL interpreter is responsible for interpreting the context inference process described using the DSL. The DSL interpreter relies on a workflow engine and on a rule engine for executing the instructions written in the DSL specification. The run-time interpretation establishes the relation between each section of the DSL specification and specific elements of the middleware. The control interface

supports the interaction of the context inference manager with the resource manager and the context modules. The context modules represent the method implementations necessary for the inference. Adaptations are triggered as the context inference manager monitors the status of specific resources through the resource manager, which supplies a proxy to locate and acquire data from both internal and external sensors of the mobile device (e.g., accelerometer, battery, memory allocated, CPU load). The resource manager is also responsible for cleaning, correcting and preprocessing raw sensor data using a pool of methods (e.g., average over a stream of values) to allow conversion, (de)compression, (un)marshaling of data, etc.

IV. Specification DSL

When developing context-aware applications, developers have to deal with context-related issues in addition to the main application logic. Often, context-related code, mainly for context inference, is entangled with application code, complicating further development and maintenance. Moreover, if the developer is aiming at adaptation of the context inference process, this requires a significant amount of conditional, event-based and periodic coding to be included.

With a DSL it is possible to mitigate these problems and easily express computations in the context-awareness domain, allowing a concise, more intuitive, specification of resources, inference behavior and their processes. Our main goal in designing the DSL was to create an appropriate set of abstractions that reduced the development effort for the programmer when performing tasks associated with context inference. We define a set of high-level abstractions for looping and periodic tasks, asynchronous event handling, condition testing, and rule declarations, among others. A dedicated specification of context inference allows a more powerful mechanism to define inferences and most importantly their adaptation rules, as opposed to a external libraries or APIs. The information specified in the DSL thus allows rapid prototyping of adaptable context inference processes, flexible adaptable behavior management and clear evaluation of conflicting rules.

A. Language Features and Requirements

A context inference process transforms raw sensor data into useful context information. Often, this process is defined statically and thus exhibits a fixed behavior. However, the inference of context information should be dynamic, mainly because of the volatile nature of context and the varying operating conditions of mobile devices. Fig. 4 depicts an overview of a generalized context inference process, which considers three main stages: *data acquisition*, *data processing*, and *context identification*. Each stage embodies methods to accomplish a task, which expose parameters that can be used to configure the process. Changes to this process accomplish the context identification with a different impact on several metrics such as execution time, energy, power and CPU load.

The DSL concept focuses on the development of context inference processes. The domain-specificity allows tailored high-level abstractions to be composed in a structure that

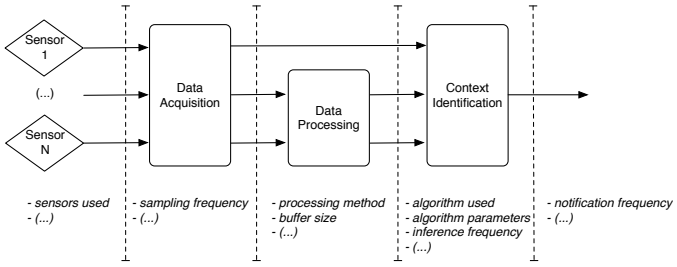

Fig. 4. Generalized context inference workflow diagram.

allows the description of the inference process, which in another general-purpose language would require extensive programming to accomplish. Our DSL allows the specification of the most important components of the context inference process within a single description by primarily specifying: (1) sensors and algorithms used for inference; (2) parameters used for adaptation; (3) structuring of the inference process into execution blocks; and (4) a set of priority-organized rules for adaptation to certain conditions and events.

B. Language Structure

In the DSL, a context inference process and its adaptable behavior is specified by three main sections: *declarations*, *operations*, and *rules* (see DSL component in Fig. 2).

Declarations are the initial DSL structural section that declares necessary variables, methods, and default parameter values that are used within the operations section (Fig. 3, line 1). The role of this section is to associate the DSL to the middleware by bridging DSL elements to middleware components. Mainly, the association is established by the match of methods whose respective implementations exist in context module components of the middleware. Such implementations of methods for inference are supplied by the application developer. For example, in line 1 of Fig. 3, the infAct inference method used within the DSL is implemented by the middleware's context module fftKnnInf. This implementation is further configured with a default initial value for FftNrSamples parameter. The inference method also specifies the output activity contexts IDLE, WALK, and JOG.

The operations section is responsible for specifying the main workflow for the inference process. It defines the necessary sequential steps that identify the context information, which can include several layers of inference and specific inference application cases. This section is built with a main execution block, and subsequent sub-block structures that can be defined to frame specific areas of the inference workflow. Such sub-block structures are used to concentrate operation steps that may be activated or deactivated, allowing a more dynamic structure. Execution blocks can be associated with execution properties such as the inference periodicy. Lines 3 to 6 of Fig. 3 define a RUN block corresponding to the operations section, parameterized to execute with a 1 second period. In this example, an output context variable is defined as the result of the inference method being applied.

The rules section specifies the adaptation strategies that are applied when triggering events or valid condition testing occurs. Common triggering conditions are memory, CPU, energy, sensor status, and context, which are available for use within the DSL. The adaptations are described by means of rules that adjust the behavior of the inference process specified within the operations section. Rules can be assigned an order that accommodate several sequential adaptation behaviors, e.g., when battery life starts decreasing one might start by adjusting some technique parameters, and only if battery life decreases to an even lower level then one might replace one technique for another. The existence of multiple rule blocks assigned to different conditions or events could cause conflicts, as incompatible actions could be enforced if multiple rules where activated simultaneously. Conflicts are solved by prioritization, where rule blocks are prioritized by their order of specification (in Fig. 3, Rule A has higher priority than Rule B). At any given moment, only one rule block can be activated, being the highest priority rule block where the triggering condition is valid. If a rule is applied and afterwards its triggering condition is no longer valid, the context inference manager performs a rollback on the executed rule and thus on the previous executed adaptation, returning to the default configuration. Lines 8 to 15 of Fig. 3 define a rules section composed of two rule blocks, one for a energy and CPU event and another for a condition test on the inference elapsed time. Rule management is conducted solely in this section and thus adding, removing or editing rules can be accomplished without an added programming overhead.

C. Application Scenarios

In this section we present another motivational DSL specification example considering a scenario of a location-based context inference system. A growing number of mobile phone applications are location-based systems that often require GPS capabilities for location context information. Unfortunately, GPS incurs an unacceptable energy cost and it may even not be available at times. Alternatives sensors (e.g., WiFi, GSM) for location are required when energy needs to be saved (e.g., when the phone is running low on battery) or when GPS is not available, even if it incurs on the loss of some localization accuracy (e.g., [5]). Fig. 5 depicts a DSL specification for this scenario, where a strategy for sensor change according to battery level or sensor unavailability is enforced. Line 1 declares the inference function infPos implemented by the context module positionInf with the default sensor set to GPS. Lines 3–5 define the context as a result from the output of the infPos method. Lines 7–20 define the adaptability rules, defined in order of priority: (1) every 5 minutes a test condition is evaluated to verify if the device's energy level (battery) is below 50%, in order to switch the sensor used for inference to the GSM sensor; (2) if the GPS sensor is unavailable then a switch to WiFi sensor is performed; (3) if the WiFi sensor is unavailable, then a switch to GSM sensor is enforced.

```
 1: infPos IS positionInf(sensor=SENSOR.GPS){position};
 2:
 3: RUN[period=5sec]{
 4:    context = infPos();
 5: }
 6:
 7: RULES{
 8:    EVERY 5min: ENERGY.LEVEL < 50%
 9:                && SENSOR.GSM.AVAILABLE{
10:       infPos IS positionInf(sensor=SENSOR.GSM);
11:    }
12:    EVENT: SENSOR.GPS.UNAVAILABLE
13:           && SENSOR.WiFi.AVAILABLE{
14:       infPos IS positionInf(sensor=SENSOR.WiFi);
15:    }
16:    EVENT: SENSOR.WiFi.UNAVAILABLE
17:           && SENSOR.GSM.AVAILABLE{
18:       infPos IS positionInf(sensor=SENSOR.GSM);
19:    }
20: }
```

Fig. 5. DSL specification for a location context inference process adaptable to sensor availability and device energy.

V. Case Study Application

This section presents a case study for experimental evaluation of the proposed approach. The case study consists in a context-aware application that is present in a smartphone and infers the user's physical activity through acceleration data (*standing*, *normal walking*, *race walking* and *running*). The inference process is adaptable being composed of a two-level hierarchy and adjustable in the inference period and algorithms, due to energy and CPU concerns. The use of a hierarchical inference scheme allows for a refinement of the computational complexity associated with context inference. Algorithms that are specific to certain contexts relieve the application from having to use a single more sophisticated algorithm in order to distinguish all necessary contexts. This adaptation ultimately allows for lower CPU load, thus lowering the inference execution time and energy consumption. In addition, an increased inference period means a lower number of inferences computed and thus a reduction of the overall CPU load compounding the overall energy reduction of the inference process and thus of the aggregate application. On the down side, however, increasing the inference period may cause contexts to be detected with some delay or even be missed altogether. Nonetheless, it may also allow skipping over some momentaneous, erroneous context inferences.

A. Implementation

In this experimental implementation, the goal is to provide the context inference with a behavior that identifies physical activities as accurately as possible while minimizing the energy and CPU load by adjusting the inference period and method. This adaptable behavior is summarized as follows: (1) initially the context inference is computed with a small inference period of 1 second; (2) for context identification a first level inference method is used (inference1 – signal differences and threshold classifier) to distinguish between *standing*, *walking* and *running* contexts; (3) in the presence of the *walking* context, a more complex method is used

(inference2 – frequency signal analysis and nearest neighbor classifier) to further distinguish between *walking* pace contexts; (4) whenever the inferred context remains constant for at least the last three inferences, the inference period is increased to 2 seconds; (5) as the battery level of the device diminishes, the period is further increased and the more complex method is reconfigured with a smaller processing window size parameter; (6) if the CPU reaches high load, then the inference level for identifying the *walking* paces is deactivated; (7) if no rule is applied, the inference is conducted at its default implementation state.

The DSL code for this case study is depicted in Fig. 6. Lines 1 and 2 declare the two inference methods as well as a default parameter value. The declared inference methods must be recognized by the middleware. Lines 4–12 define the operations section where the main inference loop is executed. In this structural section, two operational blocks are defined: level1 and level2. The two blocks will allow the activation and deactivation of the two-level inference hierarchy defined. Lines 14–33 specify the adaptation rules. There are four rules defined by priority, one rule is history-related (Rule A), two rules are energy-related (Rule B and C) and one rule is CPU-related (Rule D). The first rule captures the condition regarding three consecutive identically inferred contexts and specifies a corresponding increase of the inference period. The subsequent energy rules define that below a specific battery level, adaptations need to be performed. For a battery level below 60% only the period is increased, but in contrast, for a battery level below 30% the period is increased even more and the inference2 method is reconfigured with a different fftWindow parameter value. The CPU rule defines that above an 80% CPU load the operations block level2 is deactivated, i.e., no longer being computed.

```
 1: inference1 IS inference1{STANDING, WALKING, RUNNING};
 2: inference2 IS inference2(fftWindow=512){NORMAL_WALKING,
  |                                         RACE_WALKING};
 3:
 4: RUN[period=1sec]{
 5:    level1{
 6:       context=inference1();
 7:    }
 8:    level2{
 9:       IF(context == WALKING)
10:       THEN context=inference2();
11:    }
12: }
13:
14: RULES{
15:    // Rule A
16:    EVERY RUN: CONTEXT.HISTORY.EQUALS(3){
17:       RUN.period=2sec;
18:    }
20:    // Rule B
21:    EVERY 5min: ENERGY.LEVEL < 60% &&
  |                 ENERGY.LEVEL > 30%{
22:       RUN.period=3sec;
23:    }
24:    // Rule C
25:    EVERY 5min: ENERGY.LEVEL ≤ 30%{
26:       RUN.period=5sec;
27:       inference2 IS inference2(fftWindow=256);
28:    }
29:    // Rule D
30:    EVERY RUN: CPU.LOAD > 80%{
31:       level2.off();
32:    }
33: }
```

Fig. 6. DSL code for the adaptable activity context inference.

TABLE I
INFERENCE COMPUTATIONAL IMPACT AND OVERALL ACCURACY.

Inference	Accuracy (%)	CPU Load (%)	Power Cons. (mW)
Default	92.2	24.3	244
Rule A	92.4	12.9	167
Rule B	93.4	8.9	148
Rule C	79.5	5.5	129
Rule D	100.0	3.1	123

B. Results and Discussion

The case study highlights some key characteristics of the DSL and middleware approach described in this paper. The DSL includes simple, yet powerful elements, that allow complex behavior to be defined in a flexible and agile format without troublesome programming. As a comparison, we implemented the same inference behavior using J2ME since it is a very common development language for smartphone applications. The J2ME implementation required 250 lines of Java code. In contrast, our DSL implementation uses only 33 lines of code, a reduction of more than 85%. More importantly, our DSL specification translates in a substantial reduction of code complexity. For example, adding a new rule would amount to additional boilerplate code in J2ME, whereas in contrast in our DSL specification such addition would consist only in an additional rule block description.

Overall, using the DSL approach described here relieves the programmer from a wide range of conceptual and practical tasks namely: (1) creating appropriate mechanisms for the scheduling of inferences; (2) guaranteeing mechanisms to access information on system resources, i.e., battery level, power consumption and CPU load; (3) creating listener interfaces for the resources to process and dispatch events such as low energy or high CPU load; (4) dealing with synchronization issues between the looping inference calls and the resource events; (5) use of extra conditional statements that would adapt the inference method calls according to resource events.

Considering the computational impact and also the accuracy obtained using such context inference process, summarized in Table I are results of the system execution, during 10 minutes, over sets of random acceleration data samples of all activities of interest. The accuracy provided is stable in most rules, except on Rule C where the frequency analysis algorithm is performed over a smaller window of number of samples. Accuracy is higher in Rule D as it disables the inference of the detailed *walking* contexts and thus only requires the implementation to distinguish between three contexts. Regarding power consumption and CPU load, as the rules A through D reduce complexity of the inference by incrementing the period, changing algorithm parameters and deactivating inference levels leads to a decrease in the operational metrics.

VI. Related Work

Several approaches for context-aware application development have been proposed, due to the increasing importance of context-awareness, and also by the demand for solutions to facilitate this complex type of development. Approaches spawn mainly in the form of frameworks (e.g., JCAF [6]), middleware (e.g., MobiPADS [7], MidCASE [8], CARISMA [9]), and languages (e.g., ContextL [10], Subjective-C [11], EventCJ [12]).

Although these approaches have improved the overall development of context-aware applications, to best of our knowledge they have not specifically addressed the specification of adaptable context inference processes. The main concern of these approaches has been to assist the use of context by the applications. As example, MobiPADS [7] and CARISMA [9] do not address how a context is obtained, yet contexts are used to adapt services provided to applications. MobiPADS additionally relies on an XML-based language for service interaction specification, supporting the definition of context events, but without addressing how they are computed. Our solution mainly distinguishes itself from previous approaches by focusing on the specification of context inference processes and their management, considering system properties and application requirements. We thus allow context-aware application development based on a flexible programming approach for control of dynamic run-time adaptable context inference. Additionally, our context-oriented DSL does not extend some other host general-purpose language. It is independent from any underlying language and device platform, therefore depending solely on the middleware infrastructure for execution. In fact, context-oriented programming (COP) concepts have commonly been put into practice as extensions to several languages, each one with its own approach to the COP paradigm and with implementations suffering in general from a large execution overhead [13]. ContextL [10], Subjective-C [11], and EventCJ [12] are interesting examples of COP languages that modify the behavior of a program by associating code definitions with context-related layers that are activated according to the current context. Still, no focus is given to context inference.

There have been, however, some approaches to context inference adaptability due to the problematic energy limitation of mobile devices, whose battery is always finite. Energy-based adaptation has been used to switch sensors in location-based systems, where different sensors provide different position accuracies but also with different power consumptions (e.g., [5]). Unfortunately, the adaptable behavior specified is application-specific and has not been addressed aiming at a general and reusable approach.

VII. Conclusions

The increasing demand of context-aware applications and services and the computational constraints of mobile devices require highly flexible and adaptable context inference approaches. In this paper we presented a domain-specific language (DSL) and middleware approach for the specification and execution of dynamically adaptable context inference processes. Our approach aims at reducing the context-oriented programming effort by providing a high-level DSL that abstracts common concepts necessary for context inference, as well as information on the device operating conditions, supported by a modular middleware infrastructure. Thus, developers will be able to define a wide range of adaptable inference systems for mobile devices, with an approach that reduces the development complexity, takes into account the operating challenges in mobile platforms, and leverages adaptability in order to optimize the context inference process.

Acknowledgments

The research work presented was partially supported by *Fundação para a Ciência e a Tecnologia* (FCT) under grant number SFRH/BD/47409/2008.

References

[1] G. D. Abowd, C. G. Atkeson, J. Hong, S. Long, R. Kooper, and M. Pinkerton, "Cyberguide: A Mobile Context-Aware Tour Guide," *Wireless Networks*, vol. 3, no. 5, pp. 421–433, 1997.

[2] N. Bricon-Souf and C. R. Newman, "Context Awareness in Health Care: A Review," *International Journal of Medical Informatics*, vol. 76, no. 1, pp. 2–12, 2007.

[3] A. C. Santos, J. M. P. Cardoso, D. R. Ferreira, P. C. Diniz, and P. Chaínho, "Providing User Context for Mobile and Social Networking Applications," *Pervasive and Mobile Computing*, vol. 6, no. 3, pp. 324–341, 2010.

[4] C. Bettini, O. Brdiczka, K. Henricksen, J. Indulska, D. Nicklas, A. Ranganathan, and D. Riboni, "A Survey of Context Modelling and Reasoning Techniques," *Pervasive and Mobile Computing*, vol. 6, no. 2, pp. 161–180, 2010.

[5] Z. Zhuang, K.-H. Kim, and J. P. Singh, "Improving Energy Efficiency of Location Sensing on Smartphones," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys '10)*. ACM, 2010, pp. 315–330.

[6] J. Bardram, "The Java Context Awareness Framework (JCAF) - A Service Infrastructure and Programming Framework for Context-Aware Applications," in *Pervasive Computing*, ser. LNCS, vol. 3468. Springer, 2005, pp. 98–115.

[7] A. T. S. Chan and S.-N. Chuang, "MobiPADS: A Reflective Middleware for Context-Aware Mobile Computing," *IEEE Transactions on Software Engineering*, vol. 29, no. 12, pp. 1072–1085, 2003.

[8] Y. Bai, H. Ji, Q. Han, J. Huang, and D. Qian, "MidCASE: A Service Oriented Middleware Enabling Context Awareness for Smart Environment," in *Proceedings of the International Conference on Multimedia and Ubiquitous Engineering (MUE '07)*. IEEE, 2007, pp. 946–951.

[9] L. Capra, W. Emmerich, and C. Mascolo, "CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications," *IEEE Transactions on Software Engineering*, vol. 29, no. 10, pp. 929–945, 2003.

[10] P. Costanza and R. Hirschfeld, "Language Constructs for Context-Oriented Programming: An Overview of ContextL," in *Proceedings of the 2005 Symposium on Dynamic Languages (DLS '05)*. ACM, 2005, pp. 1–10.

[11] S. González, N. Cardozo, K. Mens, A. Cádiz, J.-C. Libbrecht, and J. Goffaux, "Subjective–C: Bringing Context to Mobile Platform Programming," in *Proceedings of the 3rd International Conference on Software Language Engineering (SLE '10)*, ser. LNCS, vol. 6563. Springer, 2010, pp. 246–265.

[12] T. Kamina, T. Aotani, and H. Masuhara, "EventCJ: A Context-Oriented Programming Language with Declarative Event-based Context Transition," in *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD '11)*. ACM, 2011, pp. 253–264.

[13] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid, "A Comparison of Context-Oriented Programming Languages," in *Proceedings of the International Workshop on Context-Oriented Programming (COP '09), co-located with ECOOP 2009*. ACM, 2009, pp. 6:1–6:6.