Using Models at Runtime to Adapt Self-managed Agents for the IoT

Inmaculada Ayala, Jose Miguel Horcas(✉), Mercedes Amor(✉), and Lidia Fuentes

Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Andalucía Tech, Campus de Teatinos s/n, 29071 Málaga, Spain
{ayala,horcas,pinilla,lff}@lcc.uma.es

Abstract. One of the most important challenges of this decade is the Internet of Things (IoT) that pursues the integration of real-world objects in the virtual world of the Internet. One property that charac-terises IoT systems is that they have to react to variable and continuous changes. This means that IoT systems need to work as self-managed systems to effectively manage context changes. The autonomy property inherent to software agents makes them a suitable choice for developing self-managed IoT systems. By embedding agents in the devices that compose the IoT is possible to realize a decentralized system with self-management capacities. However, in this scenario new problems arise. Firstly, current agent development approaches lack mechanisms to deal with the heterogeneity present in the IoT domain. Secondly, agents must simultaneously deal with potentially conflicting changes in their behaviour, concerning self-management and application goals. In order to afford these challenges we propose to use an approach based on Dynamic Software Product Lines (D-SPL) and preference-based reasoning. The D-SPL provides to the preference-based reasoning of the agent with the necessary information to adapt its behaviour at runtime making a trade-off between the self-management of the system and the accomplishment of its application goals.

Keywords: Software Product Lines · Dynamic Software Product Lines · Goal-oriented · Internet of Things · Preference-based reasoning

1 Introduction

One of the most important challenges of this decade is the Internet of Things (IoT) [1], which pursues the integration of real-world objects in the virtual world of the Internet. One property that characterizes IoT systems is that they are composed of a globally connected, highly dynamic and interactive network of physical and virtual devices [1]. These devices have to react to variable and continuous changes in their context. This means that IoT systems need to work as self-managed systems to effectively manage context changes. With this requirement,

it is essential to have a decentralized solution to deal with the self-management of the system [4].

The autonomy property inherent to software agents makes them a suitable choice for developing self-managed IoT systems. By embedding agents in the devices that compose the IoT is possible to realize a decentralized system with self-management capacities. In fact, the notion of environment (i.e. the context) is a key concept in agent and Multi-Agent Systems (MASs) technology, since it strongly affects agent behavior [23,26]. The proactive and autonomous behavior of agents, usually modeled in terms of goals, means that they are able to be aware of and adapt to the particular context in which they are embedded according to a set of self-management goals. These goals endow agents in IoT systems with the ability to run continuously under different conditions such as changing environments, partial subsystem failures, and changing user needs. Often, they must run unattended with no interruption. Agents running in smart devices also need to adapt their overall system behavior to energy levels and varying quality in network connection. However, in this scenario new challenges arise. Firstly, current agent development approaches lack mechanisms to deal with the variability present in the IoT domain. Secondly, as part of self-management, agents must be able to dynamically adapt their goal-driven behaviour influenced by the current context situation, even at runtime.

Managing variability can be done during the phase of analysis and design. In a previous work, the variability imposed by IoT domain was done at the stage of analysis and design of the MASs for the IoT [3]. This solution involved using a Software Product Line (SPL) process [20] to model the variability of the IoT domain, the agent context, and its dependencies with agent internal behavior specified by means of goals. In SPLs, a variability model allows specifying commonality and variability amongst a set of similar products that are part of the same product family (seen as a collection of similar software systems derived from a shared set of software assets using a common means of production process). Therefore, variability models are a natural and suitable way to easily model context variability and their interdependencies with the agent behavior in agent-based IoT systems. This information is valuable for the self-management of agents at runtime, however SPL approaches are restricted to the development phase of the system. One solution is to foresee all the functionality or management an agent in an IoT system may require and include them in the SPL. The issue is to consider all possible variations and in addition, to reason about them at runtime. This would be unaffordable in time and computational resources for an agent embedded in a lightweight device.

Therefore, there is a need to produce software agents for the IoT capable of evolving and adapting to different system management requirements while meeting the application goal they were intended for. In order to meet this challenge we propose to use an approach based on Dynamic SPL (D-SPL) [10,11]. D-SPL is an area of research that applies the ideas of the SPL like variability modeling and automatic product derivation to the runtime. This makes to produce software capable of adapting to context variations and evolving resource constraints.

In D-SPLs, monitoring the current situation and controlling the adaptation are central tasks. The D-SPL provides the agent with the necessary information to evolve self-management at runtime taking into account how it affects agent goals. In this work we propose to use D-SPLs to adapt the behaviour of the agent. Our goal is to achieve a trade-off between application specific goals and self-management goals ensuring the agent preserves an acceptable quality of service for the IoT system. Such quality is quantified in terms of the *wellness* and *usefulness* of the agent at one point of its execution. These metrics, which are inferred from the variability model of the D-SPL, are used to select the appropriate plan for a given goal according to the current state of the agent. The wellness of the agent is defined as a general condition of the agent in terms of its internal state (such as available resources, and activated goals, scheduled plans). The usefulness is measured in terms of the goals that the agent potentially can bring about and the goals that are currently maintaining. The scope of our approach is intented to specific closed IoT scenarios where agents reasoning only takes into account the monitorized environment values and agents goals do not interfere with other agents in the MAS.

This paper is organized as follows: Sect. 2 overviews our approach. Section 3 presents our case study and provides background in variability modeling and related work. Section 4 explains our mechanism to adapt agent behavior at runtime. This mechanism is validated in Sect. 5. The paper closes with the conclusions and future work.

2 Our Approach

We propose to develop IoT applications as a population of agents embedded in IoT nodes, that interact with each other, and which are self-managed. In emerging domains such as the IoT, software is becoming increasingly complex with an extensive variation in both requirements and resource constraints. Developers are pressed to deliver high-quality software with additional functionality, on short deadlines, and more economically. In addition, IoT environments demand a higher degree of adaptability from their software systems. Computing environments, user requirements, networking and interface mechanisms between software and hardware devices such as sensors can change even at runtime. In order for agents to be embedded in devices of the IoT, while maintaining the decentralization of the self-management, the variability and self-management must be handled at the agent level. The proactive and autonomous behavior of agents, usually modeled in terms of goals, enable agents to be aware of and adapt to the particular context in which they are embedded according to a set of self-management goals. Until now, different agent technologies have been adapted or extended to provide support for some devices of the IoT (mainly sensor motes and mobile phones) [2]. In a previous contribution, we have presented an SPL process for the development of self-managed agent-based systems for the IoT that considers these issues [3]. In SPL approaches, variability is bound at development time. However, they do not support the dynamic reconfiguration of agent

architectures enabling the agent self-management at runtime. In this work, the variability model is used also at runtime to drive the adaptation of the agent to changes in the environment.

In D-SPLs, monitoring the current situation and controlling the adaptation are central tasks. These tasks are activities of the commonly known as MAPE-K loop (Monitoring, Analyzing, Planning and Execution - Knowledge) of autonomous (or autonomic) systems. The agent realizes the MAPE-K cycle, qualifying the agent it-self as being an autonomous system. Because the variability model is the core artifact for guiding system adaptation, the agent must be able to consult (in one form or another) the runtime variability model to identify adaptations. Then the variability model is the typical knowledge part of a MAPE-K feedback loop. Here, we focus on heterogeneous agents at runtime and how they deal with the dynamic adaptation of their behaviour due to changes in environment according to a self-management policy using the variability model. Our approach is depicted in Fig. 1.

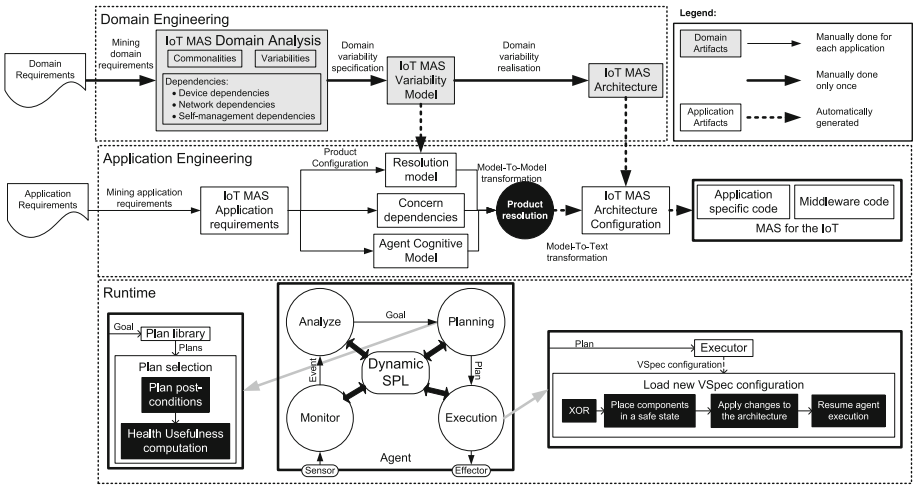

Fig. 1. SPL process for self-managed agents in the IoT.

The SPL employs two-life-cycle approach that separates domain and application engineering. While the first part, the Domain Engineering, concerns with analyzing MASs for IoT as a product line in order to produce any common (and reusable) variable parts, the second part, the Application Engineering, involves creating product-specific parts and integrating all aspects of individual products [11]. Our approach uses the information provided in the Application Engineering process to deal with the adaptation of agent behaviour using models at runtime. One of the latest steps of the Application Engineering considers the modeling of the agent cognitive concepts using CVL [12] (Agent cognitive Model box in Fig. 1). This model is used to obtain the application architecture (IoT MAS

Architecture Configuration box in Fig. 1). The MAS application architecture configuration describes the architectures of the agents of the IoT system.

As part of the Application Engineering process, the self-management policies of each agent are also selected. A self-management policy consists of a set of self-management goals, which are achieved by a set of self-management plans. The configuration of these self-management policies takes into consideration the architectural dependencies of the agent and application requirements. Plans for self-management are generated as part of the Model-To-Model (M2M) transformation that generates the application architecture. Agents continuously monitor the environment as part of their self-managed behaviour. When a change in the environment occurs, an event is thrown. These events are analysed to activate new goals that manage the new context. The agent, as part of its planning task, selects a plan to achieve the activated goal. The plan contains a set of actions that can enable or disable specific components or to change its configuration. At this point of the execution the architecture configuration is dynamically adapted. Such adaptation is addressed using an approach based on preference-based reasoning [18,25]. Self-management for lightweight devices usually relies on common policies that can be combined in order to adapt the agent and the device where it is embedded. Agent developers must benefit from the reuse and combination of previously defined self-management policies taking into account the requirements of the application and the dependencies with the agent architecture. Changes in the agent behaviour are required when the environment changes and the self-management policy being applied adapts to the variation. The application of a self-management policy carries out the activation of a set of self-management goals, and the selection of plans to achieve both application and self-management goals are influenced by this change. As part of self-management, the agent must be able to deal with adaptation dynamically and in an autonomous manner. The ideal solution is to make a trade-off between these goals, ensuring the achievement of agent application goals, while maintaining the agent in the best conditions as long as possible with an acceptable quality of service whenever possible.

At runtime, in order to implement the aforementioned trade-off, the activation of self-management goals is driven by the *wellness* and *usefulness* of the agent. These properties quantify different concerns of the agent. Wellness is concerned with the quality or state of being healthy. The wellness of the agent is defined as a general condition of the agent in terms of its internal state (such as available resources, and activated goals, scheduled plans). More specifically wellness is related to self-management policies that are contained in its architecture. The usefulness is measured in terms of the goals that the agent potentially can bring about and the goals that are currently maintaining. In general (but not in all cases), plans that increase or emphasize agent usefulness are going to erode its wellness. Both metrics are inferred from the current configuration of the agent architecture and are used in the selection of plans. Then plans are tagged by their contribution to agent wellness and usefulness. The reasoning mechanism of our agent chooses plans to achieve its goals taking into account these factors.

When wellness factor is good, the agent tends to choose those plans (to achieve application and self-management goals) that increase its usefulness in spite of its well-being. When the wellness of the agent gets worse, it behaves conservatively to maintain its current state although its usefulness decreases.

3 Background

This section presents the case study that will be used to illustrate our proposal, additionally provides the background in CVL and related work.

3.1 Case Study

In order to illustrate our proposal, we use, as a case study a smart shopping centre. The case study focus in a shop endowed with IoT devices intended to enhance the shopping experience.

This system uses different technologies to improve the shopping experience of customers through an *active environment*. To become an active environment, the physical space is endowed with a set of small devices, namely beacons [27], that can send signals to smartphones and other personal devices entering their immediate vicinity. Signals, which contain information about the context, are sent via Bluetooth Low Energy Technology (BLE for short) [9]. BLE is a part of the Bluetooth 4.0 specification released back in 2010. It has a different set of protocols from "classic" Bluetooth, and devices are not backwards-compatible. Accordingly, you can now encounter three types of Bluetooth support: Bluetooth (devices supporting only the "classic mode"; Bluetooth Smart Ready (devices supporting both "classic" and LE modes); and Bluetooth Smart (devices supporting only the LE mode). Beaconing can be applied in all kinds of valuable ways. To put it simpler, beacons transmit data to devices that are in range to enable transmission (immediate, near or far), allow the user to be located in places where GPS is not as useful, for example, in a museum, park or shopping centre. For shopping centres in particular, beacons are important because they allow a more precise targeting of customers in a premises. For example, a customer approaching a store, could receive a message from a battery-powered beacon installed there, offering information or a promotion that relates specifically to products displayed there. In a different area or location of the same store, another beacon transmits a different message. Before beacons, marketers used geofencing technology, so that a message, advertisement, or coupon could be sent to consumers when they were within a certain range of a geofenced area, such as within a one-block radius of a store. However, that technology typically relies on GPS tracking, which only works well outside the store. With beaconing, marketers can lead and direct customers to specific areas and products within a particular store or the shopping centre itself.

Our case study focuses on a single store (see Fig. 2), BLE Beacons are scattered and spread over the shop space, linked to the different furniture elements. These elements contain are associated with specific categories of products and

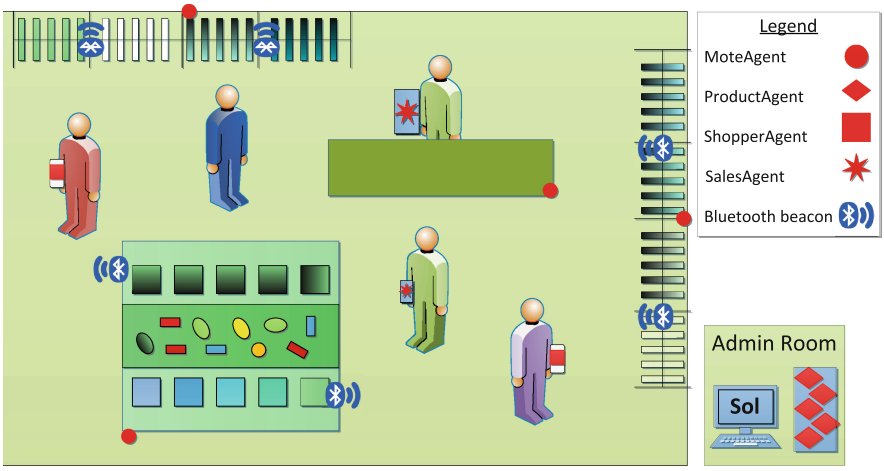

Fig. 2. Overview of the multi-agent system of the smart shopping centre.

special offers. The information loaded in each BLE Beacon is related to this category of product or special offer. Then, when a customer with a smartphone is close to a beacon (and also a category of products or a special offer), the device receives a (context-aware) recommendation. The salespeople also benefit from the technology around them. Store employees are informed of the position of customers in the store, and receive information about the environmental conditions of the shop in real time on their hand-held devices. This system has been designed as a MAS composed of agents embedded in the different devices that comprise the application. There are agents embedded in sensor motes (which provide the environmental conditions of the store), in the personal devices of the users (both customer and employees) and in the shop manager's computer. Henceforth, we use *ShopperAgent* to illustrate our approach.

3.2 The Common Variability Language

Variability of SPLs can be specified using different modeling languages. Although feature models [15] have been very popular in the SPL community over the last decade, recently the Common Variability Language (CVL) [12] has been proposed as a standard. Both variability languages can be used in our proposal, but here we opt to use CVL.

CVL is a domain-independent language for specifying and resolving variability over any instance of any language defined using a MOF-based metamodel (e.g. UML) which is called the *base model*. In a CVL process (see Fig. 3) all the variations of the *base model* are specified using the *variability model*. Each possible variation expressed in the *variability model* is called a *resolution model*. When one of the *resolution models* is selected, the CVL tool is executed and obtains the *resolved model*, a product model, fully described in the MOF-based

metamodel, which is a variation of the *base model* according to the choices that have been made in the *resolution model*.

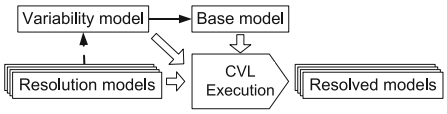

Fig. 3. Common variability language overview as specified in [12].

The *variability model* (see top of Fig. 4) is a specification in CVL of the base model variabilities and their relationships, and it is defined in two steps. Firstly, the *variation points* (i.e. What varies?) are marked over the *base model* (see center of Fig. 4). There are different types of variation points: to indicate the *existence* of an element; for the *substitution* of a particular part of the base model; for the *value assignment* of a particular slot of the model; or to set a domain specific variability associated with objects. Additionally, a set of variation points can be grouped in *Configurable Units*.

Secondly, to complete the *variability model* the CVL process specifies, separately from the base model, *Variability Specifications* (VSpec). These entities are specifications of abstract variability that are organised in tree structures (VSpec trees) representing logical constraints on their resolutions (see Fig. 4). The sub-tree under a VSpec means that the resolution (i.e. the selection of a particular variability decision) of this VSpec imposes certain constraints on the resolutions of the VSpecs in its sub-tree. Additionally, it is possible to specify explicit constraints that are also known as *crosstree constraints* (e.g. *not(Color)* in Fig. 4). A VSpec tree can be composed of different VSpecs that have different meanings. In the modeling that is shown in following sections we use *choices*, variability classifier (*VClassifier*), *value assignments*, *CVSpec* and *group multiplicities*. *Choices* represents yes/no decisions or features that can appear in the base model or not (e.g. *BothSides*). *VClassifiers* mean having to create instances and then providing per-instance resolution for the VSpec in its sub-tree. *Variable* requires providing a value of its specified type (e.g. *speed:Integer*). *CVSpecs* are used to encapsulate sections of the VSpec tree and their resolution requires resolving the VSpecs inside it. *Group multiplicities* are used to apply restrictions over the number of children of a VSpec that can be chosen (e.g. *Type* and its children *Color* and *BW*). Furthermore, the appearance of VSpecs in a selection can be optional (linked by dashed lines as *BothSides*) or mandatory (linked by solid lines as *Type*). Formally, a VSpec tree is defined as follows:

Definition 1. *A* ***VSpec tree*** *is a tuple* $VST = (V, E, F, G, C)$*, where* V *is a finite set of VSpecs,* $E \subseteq V \times V$ *is a set of directed child-parent edges;* F *is set of logic formulas over* V *in Conjunctive Normal Form (CNF);* $G \subseteq 2^E$ *are non-overlapping sets of edges participating in group multiplicities; and* $C : G \rightarrow N_0 \times N_0$ *is a mapping from a group to a pair denoting the cardinality of the group.*

The following well-forcedness constraints hold in VST: (i) (V,E) is a rooted tree; (ii) all edges in a group multiplicity shares the same parent, so if $g \in G$ *and* $(f_1, f_2), (f_3, f_4) \in g$ *then* $f_2 = f_4$*; and (iii)* $\forall (m, n) \in range(C), m \leq n$.

Definition 1 states that mandatory VSpecs are represented by logic formulas in the form $parent \wedge child$ (e.g. $Scanner \wedge Type$). A VSpec tree can be translated to propositional logic if we interpret VSpecs as variable names. To do so, we use the function $t(\cdot)$ and $GM_{m,n}(\dots)$. Given boolean variables $f_1, \dots, f_k$ and $0 \leq m \leq n \leq k$, $GM_{m,n}(f_1, \dots, f_k)$ holds iff at least m and at most n of $f_1, \dots, f_k$ are true. In addition, we assume that variables are logic variables that are always true. Formally, the function $t(\cdot)$ is defined as follows:

Definition 2. *For a VSpec tree* $VST = (V, E, F, G, C)$ *define:*

$$t(VST) = \bigwedge_{(c,p) \in E} (c \rightarrow p) \wedge \bigwedge_{g \in F} g \wedge \bigwedge_{\substack{g \in G, \\ (m,n) \in C(g), \\ g = (f_1, f), \dots, (f_k, f)}} (f \rightarrow GM_{(m,n)}(f_1, \dots, f_k))$$

The effect of the variability model on the base model is specified by *binding variation points* (black arrows in Fig. 4), which relates the base model, the variation points and the VSpec tree. Once the *variability model* and the *base model* have been defined, the VSpecs of the VSpec tree are resolved taking into account its specific type (e.g. *choices* are selected or not, values are given to *variable assignments*,...). As stated, this resolution of VSpecs is referred to as a *resolution model*. A *resolution model* holds dependencies entailed by the VSpec tree structure and the crosstree constraints of the *variability model*. Using Definition 2, a *resolution model* is formally defined as follows:

Definition 3. *A* ***resolution model*** $RM : V \rightarrow \{\bot, \top\}$ *is an assignment of truth values to the propositional logic formula* $t(VST)$ *that makes this formula true.*

For instance one of the resolution models of the VSpec tree of Fig. 4 assigns true to *VendingMachine*, *Type*, BW and assign 10 to *Speed*. Then, the resolved model will be composed of grey rectangles shown at the bottom of Fig. 4.

3.3 Related Work

Although SPL technology has been successfully applied to different application areas [5] that includes MAS [8,17,19], we have not found any application of D-SPL to MAS.

Regarding SPL, the integration of the two technologies is known as MAS-PL (Multi-Agent System Product Lines) and related works focused on different aspects of agent development. In [8] the Gaia methodology is modified to include

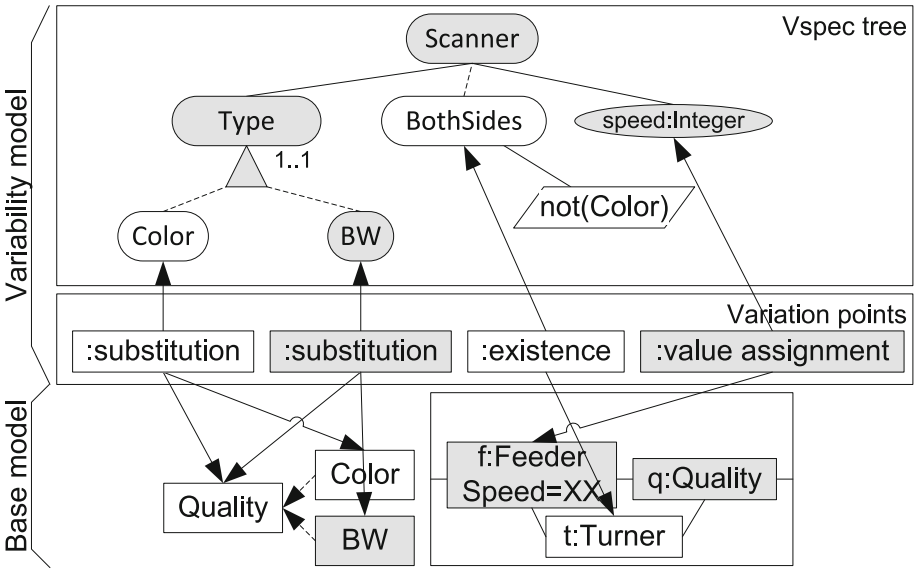

Fig. 4. Example of CVL specification extracted from [12].

SPL in the analysis and design phases of a MAS. The use of SPL allows to reduce by 48 % the design documentation time at least in the case study presented, compared to the original Gaia. MaCMAS [19] is a methodology that uses formal methods and SPL to model autonomous and self-adaptation properties of MAS. It uses SPLs to model the evolution of the system taking into account the different products contained in the SPL. The work presented in [7] focuses on the Application Engineering process by extending an existing product derivation tool for the MAS-PL domain. This proposal offers a complete SPL process with tool support to generate Jadex agents [21]. Finally, the paper [6] shows how SPL can be used to create tailor-made products based on agent platforms. However, none of these approaches consider the limitations of IoT devices where the agents are embedded.

Different works apply preference-based reasoning for controlling selection of plans in cognitive agents systems, but it has not been previously applied in the context of self-management as we do in this work [13,14,18,25]. The purpose of these works is to provide information to the agent in order to choose the appropriate plan for a given situation. In the context of the GOAL agent programming framework, this information is provided by means of linear temporal logic and is used to constraint the election of specific tasks [14]. Works [13,18] explore the use of preference in a similar way as we do in this proposal using functions that evaluate suitability of a plan. The work [13] explores the expressivity needed to specify such behaviour in the Blocks World domain, while [18] uses functions evaluated at runtime to select plan to accomplish goals with constraints. The work [25] uses summary information in order to compute preference of plans and to order the accomplishment of plan sub-goals.

In the future, the IoT will provide a large-scale environment for (intelligent) software agents. Moreover, in such open and dynamic environments, MAS need to be both self-building (able to determine the most appropriate organizational structure for the system by themselves at runtime) and adaptive (able to change this structure as their environment changes) [24]. Some works on scalability and MAS research deal with scalable architectures and applications (mostly residing in the data mining field); and algorithms and techniques for increasing scalability through change of the MAS environment.

4 Agent Adaptation by Dynamic SPL

This section explains how to compute wellness and usefulness factors that drive the agent behaviour for self-management. Wellness and usefulness are conflicting influences that a self-managed agent must take into consideration at runtime when selecting which plan is preferred to accomplish a goal. The wellness and usefuleness vary at runtime and also the goals that the agent accomplish. In order to compute these factors, we use the variability model which is included in the knowledge of the agent (see Sect. 2). The generation of this model is out of the scope of this paper and has been presented in a previous contribution [3], so, here we focus on how this model is used to compute the metrics.

As stated before, wellness describes the condition of good physical and mental health, especially when actively maintained by proper resources, activity and avoidance of risky behavior. For an agent, the function of wellness provides a quantification of the quality or state of being healthy in physical resources, especially as the result of deliberate effort. The concept of usefulness function is similar to the utility functions used in autonomic computing [16] to guide the self-management behaviour. Both functions measure the suitability of an state for an agent. However, utility focus on states that the agent wants to reach, while usefulness focus on the potential of the agent to accomplish its goals.

4.1 D-SPL of the *ShopperAgent*

The DSPL of the *ShopperAgent* (it is graphically represented in Fig. 5) contains the elements that the agent uses to interact with its user and beacons. For example, regarding the context, the value of the *Beacon received* requires that the crosstree constraint *Inside* holds. *Inside* is the child of *Shop* by means of a group multiplicity. For goals, each of these VSpecs have three children, *Activation*, *Achievement* and *Plans*, which have crosstree constraints attached that represents conditions for activation and achievement of the goals and the conjunction of plans to accomplish these goals. For example the goal *Show item information*, the constraints *Beacon received* and *Showing item information* are attached to its children. *Request text info* is an example of a plan that has a precondition that is always true.

As stated in Sect. 2, it is necessary to annotate (i.e. quantify) plans with its contributions to wellness and usefulness of the agent. The wellness is related with

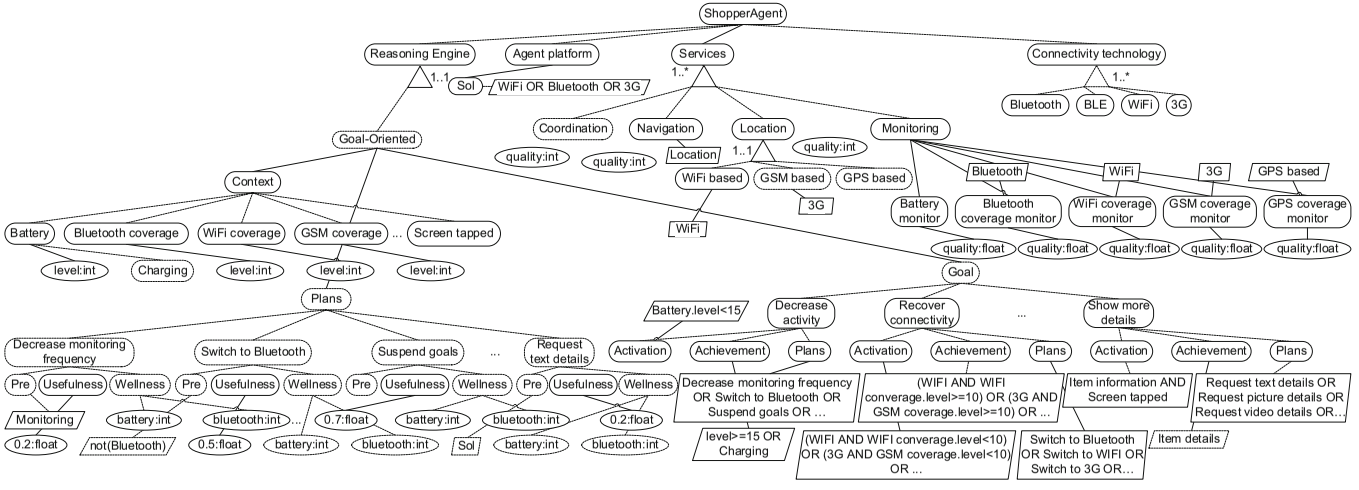

Fig. 5. Partial view of the dynamic SPL of the *ShopperAgent*.

the monitoring associated to self-management. In the case of self-management plans, the contribution to agent's wellness has been previously provided by an expert as a natural number between a range that quantifies how good or bad is the plan for agent wellness. On the other hand, the contribution of plans to agent usefulness is inferred by the resultant VSpec tree, specifically using the crosstree constraints associated with plans that state their requirements for their execution. So, if a self-management plan affects the requirement of the other plans of the agent, then it is going to negatively affect the usefulness metric. In the case of application specific plans, these values must be provided by the agent developer. Specific details of how to assess health and usefulness are provided in the following subsections. Finally, the D-SPL also includes additional information that is used for the self-management process too. Such additional information includes goals that the agent is achieving in an specific moment and the quality of the services provided.

4.2 Computation of Usefulness and Wellness

The usefulness metric captures the agent's capacity to accomplish goals and the quality of this accomplishment. This metric is computed at runtime from the information provided in the D-SPL of the agent (see Fig. 5). The D-SPL contains the number of goals that the agent can potentially accomplish in a given moment (children of VSpec *Goal* in Fig. 5) and those that are currently achieving (VSpec *Achieving*). The number of goals that the agent can accomplish could change because there is no plan that can accomplish a certain goal or they are explicitly suspended due to a self-management policy. On the other hand, children of *Services* VSpec provide a value of the quality of service of its related component in a given moment. This quality can represent an accuracy in the data provided or the frequency of a sampling or even a sum of these two values. Nonetheless, the maximum quality provided by a service is rated as 100 and the minimum is 0. To calculate agent usefulness we use (1).

$$Usefulness(A) = \sum_{i=1}^{\|Go\|} Go_i + achieving + \sum_{i=1}^{\|Ser\|} Quality(Ser_i) \tag{1}$$

The usefulness of an agent is bounded and its maximum value is denoted by $Usefulness_{max}$. Usefulness of plans considers two factors, the direct effect on the agent architecture and the *perceived usefulness* of the plan. The effects in the agent architecture are explicitly annotated in the plan as a post-condition. In the case of plans for self-management the effects can be a decrease of the quality of a specific service or the removing or addition of a feature of the system. These modifications in the agent architecture can require the addition or removing of other components. For example, if we have a plan that disables the *Bluetooth Connectivity Technology*, then *Bluetooth coverage monitor*, *Bluetooth coverage* and *Switch to Bluetooth* will be removed too (see Fig. 5).

With regard to plans that come from the cognitive model, the effect on the agent architecture is usually an increment of the number of goals that the agent

is currently achieving. On the other hand, the *perceived usefulness* is a subjective value that the developer gives to each plan for each goal that it can accomplish. For example, when the mobile phone is close to a beacon associated with a specific product of the shop, *ShopperAgent* can request information of the item from the *ProductAgent* that stands for this specific product in different formats (see Fig. 5), a piece of text (*Request text info*), a picture (*Request picture info*) or a video (*Request video info*). So, the agent developer considers that the plans with the maximum quality is *Request video info*, followed by *Request picture info* and then the *Request text info*. The value of the perceived usefulness is between 0 and 100, and it is used to compare the different plans that can be used to accomplish a specific goal. Formally, plan usefulness is defined as follows:

Definition 4. *The* ***usefulness of a plan*** P *to accomplish the goal* Go *of the agent* A *is defined as* $Usefulness(P, A, Go) = Usefulness(Conf(A, P)) + Q(P, Go)$ *where* $Conf(A, P)$ *is the VST of the agent* A *after the modifications that perform* P*, and* $Q(P, Go)$ *is how the user rates the accomplishment of the goal* G *using* P*.*

Agent wellness is based on the agent internal state, which comprises different factors that can be monitored for self-management activities in the VSpec. The *variables level* that are attached to the children of *Self-management* (see Fig. 5) are located in D-SPL of the agent. Formally, it is defined as follows:

Definition 5. *The* ***agent wellness*** *is an n-tuple* $Wellness(A) = (h_1, ..., h_n)$ *where* h_i *is the value of the* level *variable of the i-th children of the* Self-management *VSpec.*

Agent wellness evolves at runtime due to changes in the context and also in the configuration of the agent architecture. For example, if *WiFi* is disabled, then *WiFi coverage* will be disabled accordingly and this value will be not part of the agent wellness. In order to compute plan wellness, plan specifications must include the contribution of the plan to each of these values. Of course, they are estimations, as due to the heterogeneity present in the IoT domain, even for hand-held devices, it is difficult to provide an exact value. So, within the post-conditions and the quality of accomplishment, agent programmers must provide an n-tuple that represents the contribution of the plan for each element that comprise the agent wellness. As for the computation of usefulness, these values are predefined for self-management and must be provided by agent developers in plans of the cognitive agent model. Then, in execution time the effect of the plan on agent wellness is computed as the direct substraction between agent wellness and the effect of the plan in agent wellness.

Definition 6. *The* ***effect of plan*** P *on the health of agent* A *is an n-tuple* $(e_1, ..., e_n)$ *defined as* $Effect(P, A) = Health(A) - Contribution(P)$*, where* $e_i \in \mathbb{R}$ *and* $e_i > 1$*, and* $Contribution(P)$ *is the estimated effect of* P *on* $Health(A)$*.*

To compute preference, we work with $SE(P, A) = \sum_{i=1}^{\|Effect(P,A)\|} e_i$. As usefulness, since, agent health is a bounded value, it has a maximum value denoted by $Health_{max}$.

This approach restricts reasoning about temporal aspect of proposition evaluation. For instance, reasoning about the eventuality in which a shopper agent is interested in a specific good although she is not interested now is not approached. Dealing with temporality of gathered knowledge would require non-monotonicity expressive enough to reason about properties, which would make our approach too complex for the features of the scenarios considered.

4.3 Preference-Based Reasoning to Deal with Conflicts

The reasoning loop of our goal-oriented agent (see bottom of Fig. 1) follows the classical agent reasoning loop [22]. The agent reacts to changes in the environment by generating goals, when appropriate, it plans to achieve these goals and finally executes one of these plans. The main differences are in the knowledge element (in our agent is the D-SPL) and in the behavior of components for planning and execution.

The goal of the planning component is to select the plan to accomplish a goal taking into account to the trade-off between agent wellness and usefulness. The preference cuantifies the prossibility to choose the plan taking into account current agent wellness and plan usefulness. Goals and plans are ordered by the preference. The expected effect is such that when the agent has a good wellness it prefers plans that are going to increase its usefulness despite its wellness, while when is the opposite case, it will prefer plans that are less harmful for its wellness. So, the relation between usefulness, wellness and preference is similar to a negative exponential function. In order to equilibrate the weight of these factors in the preference function, the values of plan usefulness and the effect on agent health are normalized (i.e. dividing by maximum values of health and usefulness). So, $Preference$ is defined as follows:

$$Preference(P, A, Go) = \frac{Usefulness(P, A, Go)}{Usefulness_{max}} - \alpha * e^{-\frac{SE(P,A)}{Health_{max}}} \tag{2}$$

Plans are ordered using the value of the application of (2) and the plan that obtains the maximum value is selected for execution. Plans are executed in the *Executor* component (see Fig. 1) and in the case of self-management plans, it is likely that they lead to a new VSpec configuration. In this case, in order to calculate VSpecs that must be added or removed, an XOR between plan post-conditions and the D-SPL is calculated. Restrictions and dependencies of the D-SPL ensures that the architecture is always going to evolve to a correct state. Then, with the information of the binding, related components are placed in a safe state, changed and applied to the agent architecture and finally the agent execution is resumed.

5 Validation

In order to validate our proposal, we have simulated the behaviour of our preference function (see Eq. 2) for different values of wellness and usefulness. As we

stated in the introduction, our goal is to guide the election of plans to accomplish goals, so when the agent has a good wellness is likely to select plans that are going to increase its usefulness in spite of its wellness. When the health of the agent is decreased by about a certain limit, plan selection is going to take more into consideration agent wellness. So, it is necessary to check the behaviour of the preference function for different values of effect in agent wellness (see Definition 6) and obtained usefulness (see Definition def:plansusefulness). Since usefulness and wellness are bounded, we can analyze the output of the function for all possible combinations of these values.

Figures from 6 to 8 illustrates the behavior of the preference function. The *Wellness* axis illustrates the effect of a plan in agent wellness, while the *Usefulness* axis is the resultant usefulness after plan application, and the color is the output of the preference function. For different values of α, the behaviour of the preference is as we intended and there are a tradeoff between health and usefulness.

The α value enables the control of the behaviour of the preference. As it is illustrated in the figures, when this value is higher, the preference function is more similar to an exponential function. This is useful to set a threshold for the agent to start behaving in a conservative way. For example, in hand-held devices when the battery is lower than a value, the application start to decrease monitoring frequency of services.

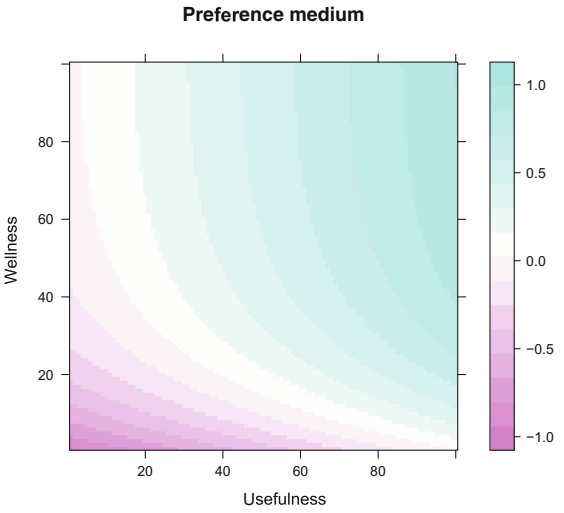

Fig. 6. Preference behaviour for a medium α.

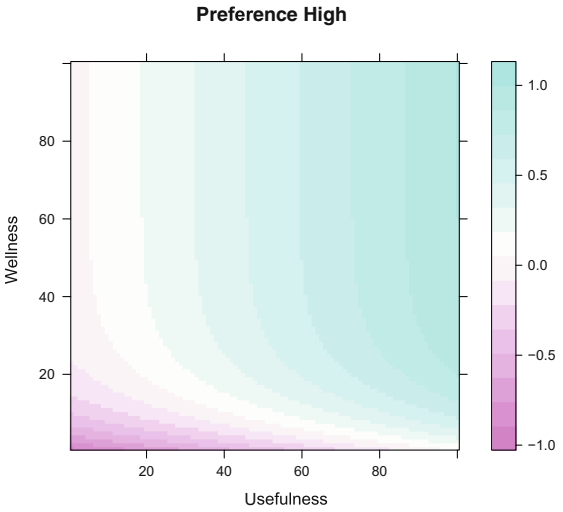

Fig. 7. Preference behaviour for high α.

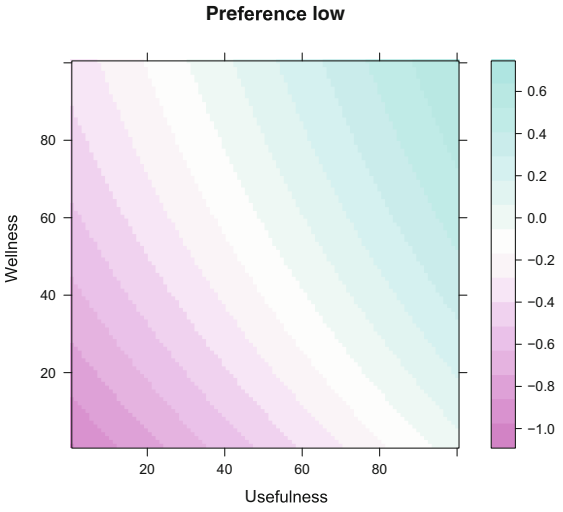

Fig. 8. Preference behaviour for low α.

6 Conclusions

In this paper we have presented an approach based on models at runtime to deal with the dynamic adaptation of agent behaviour in the IoT. In a previous contribution, we present a SPL process to generate self-managed agents in the IoT. The variability models generated in this process are used by a reasoning mechanism based on preference-based reasoning that using D-SPLs at runtime

achieves a trade-off between self-management and application goals, ensuring the agent preserves an acceptable quality of service in the IoT system. This reasoning mechanism quantifies agent *wellness* and *usefulness* at one point of its execution. These metrics, which are inferred from the D-SPL are related in a *preference* function, are used to select the appropriate plan for a given goal according to the current state of the agent. We have validated our approach showing that the proposed preference function achieves a trade-off between the effect of the plan in the agent health and its usefulness.

Currently, we are studying the behavior of MAS composed of agents with this trade-off mechanisms. Since, the accomplishment of some goals is going to be influenced by other agents, issues like negotiation will be included in our reasoning mechanism. As future work, we plan to develop a trade-off mechanism for agents with reactive reasoning engines. This mechanism will be integrated in our agents for sensor motes and mobile phones with poor computational resources.

Acknowledgements. This work is supported by the project Magic P12-TIC1814 and by the project HADAS TIN2015-64841-R (co-financed by FEDER funds).

References

1. Atzori, L., Iera, A., Morabito, G.: The internet of things: a survey. Comput. Netw. **54**(15), 2787–2805 (2010)
2. Ayala, I., Amor, M., Fuentes, L.: A model driven engineering process of platform neutral agents for ambient intelligence devices. Auton. Agent. Multi-Agent Syst. **28**(2), 214–255 (2014)
3. Ayala, I., Amor, M., Fuentes, L., Troya, J.M.: A software product line process to develop agents for the IoT. Sensors **15**(7), 15640 (2015)
4. Bono-Nuez, A., Blasco, R., Casas, R., Martín-del-Brío, B.: Ambient intelligence for quality of life assessment. J. Ambient Intell. Smart Environ. **6**(1), 57–70 (2014)
5. Bosch, J.: From software product lines to software ecosystems. In: Proceedings of SPLC, pp. 111–119. Carnegie Mellon (2009)
6. Braubach, L., Pokahr, A., Kalinowski, J., Jander, K.: Tailoring agent platforms with software product lines. In: Müller, J.P., Ketter, W., Kaminka, G., Wagner, G., Bulling, N. (eds.) MATES 2015. LNCS, vol. 9433, pp. 3–21. Springer, Heidelberg (2015). doi:10.1007/978-3-319-27343-3_1
7. Cirilo, E., Nunes, I., Kulesza, U., Lucena, C.: Automating the product derivation process of multi-agent systems product lines. J. Syst. Softw. **85**(2), 258–276 (2012). Special issue with selected papers from the 23rd Brazilian Symposium on Software Engineering
8. Dehlinger, J., Lutz, R.R.: Gaia-PL: a product line engineering approach for efficiently designing multiagent systems. ACM Trans. Softw. Eng. Methodol. **20**(4), 17:1–17:27 (2011)
9. Bluetooth Special Interest Group: Bluetooth low energy 4.1. https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=282159
10. Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic software product lines. Computer **41**(4), 93–95 (2008). http://dx.doi.org/10.1109/MC.2008.123
11. Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic software product lines. In: Capilla, R., Bosch, J., Kang, K.-C. (eds.) Systems and Software Variability Management, pp. 253–260. Springer, Heidelberg (2013)

12. Haugen, O.: Common variability language. Technical report ad/2012-08-05, Object Management Group, August 2012
13. Hindriks, K.V., Jonker, C.M., Pasman, W.: Exploring heuristic action selection in agent programming. In: Hindriks, K.V., Pokahr, A., Sardina, S. (eds.) ProMAS 2008. LNCS, vol. 5442, pp. 24–39. Springer, Heidelberg (2009)
14. Hindriks, K.V., van Riemsdijk, M.B.: Using temporal logic to integrate goals and qualitative preferences into agent programming. In: Baldoni, M., Son, T.C., van Riemsdijk, M.B., Winikoff, M. (eds.) DALT 2008. LNCS (LNAI), vol. 5397, pp. 215–232. Springer, Heidelberg (2009)
15. Kang, K.C., Lee, J., Donohoe, P.: Feature-oriented product line engineering. IEEE Softw. **19**(4), 58–65 (2002)
16. Kephart, J., Walsh, W.: An artificial intelligence perspective on autonomic computing policies. In: IEEE POLICY, pp. 3–12, June 2004
17. Nunes, I., Lucena, C.J.P., Kulesza, U., Nunes, C.: On the development of multi-agent systems product lines: a domain engineering process. In: Gomez-Sanz, J.J. (ed.) AOSE 2009. LNCS, vol. 6038, pp. 125–139. Springer, Heidelberg (2011)
18. Padgham, L., Singh, D.: Situational preferences for BDI plans. In: Proceedings of AAMAS, pp. 1013–1020. IFAAMAS (2013)
19. Peña, J., Rouff, C.A., Hinchey, M., Ruiz-Cortés, A.: Modeling NASA swarm-based systems: using agent-oriented software engineering and formal methods. SoSyM **10**(1), 55–62 (2011)
20. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques, 1st edn. Springer, Heidelberg (2005)
21. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: a BDI reasoning engine. In: Bordini, R.H., Dastani, M., Dix, J., Seghrouchni, A.E.F. (eds.) Multi-Agent Programming: Languages, Platforms and Applications, pp. 149–174. Springer, Boston (2005)
22. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 3rd edn. Prentice Hall Press, Upper Saddle River (2009)
23. Sadri, F.: Ambient intelligence: a survey. ACM Comput. Surv. **43**(4), 3601–3666 (2011)
24. Turner, P.J., Jennings, N.R.: Improving the scalability of multi-agent systems. In: Wagner, T.A., Rana, O.F. (eds.) AA-WS 2000. LNCS (LNAI), vol. 1887, p. 246. Springer, Heidelberg (2001)
25. Visser, S., Thangarajah, J., Harland, J., Dignum, F.: Preference-based reasoning in BDI agent systems. Auton. Agent. Multi-Agent Syst. **30**(2), 291–330 (2016)
26. Weyns, D., Helleboogh, A., Holvoet, T., Schumacher, M.: The agent environment in multi-agent systems: a middleware perspective. Multiagent Grid Syst. **5**(1), 93–108 (2009)
27. Wikimedia Foundation, Inc.: ibeacon. http://en.wikipedia.org/wiki/IBeacon