A goal-driven software product line approach for evolving multi-agent systems in the Internet of Things

Inmaculada Ayala *, Mercedes Amor, Jose-Miguel Horcas, Lidia Fuentes

CAOSD Group, Universidad de Málaga, Andalucía Tech, Spain

ABSTRACT

Multi-agent systems have proved to be a suitable technology for developing self-adaptive Internet of Things (IoT) systems, able to make the most appropriate decisions to address unexpected situations. This leads to new opportunities to use multi-agent technologies to develop all kinds of cyber–physical systems, which usually encompass a high diversity of devices (e.g., new home appliances). The heterogeneity of devices and the high diversity of the available technology, demand the explicit modeling of all kinds of variability for ultra-large systems. However, multi-agent systems lack mechanisms to effectively deal with the different degrees of variability present in these kinds of systems. Software Product Line (SPL) technologies, including variability models, have been successfully applied to different domains to explicitly model variability in hardware, system requirements or user-intended goals. In addition, current market trends are unpredictable, imposing novel technologies, new requirements and goals that must be incorporated immediately into the running systems without damaging them. In this paper, we combine goal-driven and SPL approaches to develop and drive the evolution of multi-agent systems in the context of cyber–physical systems. We propose an SPL process and an evolution process that define a set of models (iStar 2.0 for goals and CVL models for variability) and algorithms to automatically propagate changes to agents running in multiple heterogeneous devices, each of them with a different configuration. We illustrate the proposal in the context of a home energy management system. Finally, we have tested the scalability and performance of the proposal using randomly generated models. The results show that with our approach it is possible to manage huge iStar models of 10000 elements in seconds.

1. Introduction

The strategy *smart anything everywhere* (SAE[1]) drives the next wave of devices with electronic components inside, ranging from mobile devices, smart home appliances and a myriad of *Internet of Things* (IoT) devices that can be exploited by a new generation of *Cyber–Physical Systems* (CPS). This leads to new opportunities to build more sophisticated software products with an increasing awareness of the surrounding environment [1,2]. Indeed, the increasing complexity of current CPSs requires software systems with a higher level of pervasiveness, proactiveness and self-adaptation to changing environments. *Multi-Agent Systems* (MAS)

embedded in networked IoT devices have proved to be a suitable technology for developing self-adaptive CPSs, able to make the most convenient decision to face unexpected situations [3,4].

The growing interest in new digital businesses encouraged by the Digital Single Market initiative, brings new possibilities to use MAS technologies to develop any kind of CPS, which range from small-scale systems to large system of systems. The versatility of these systems largely depends on the level of integration and management of any kind of device that can produce relevant information. Indeed, the list of devices that can connect to the Internet continues to grow (e.g., new smart home appliances), so system development is currently facing a diversity explosion with respect to the types of devices that can be integrated. The large number and heterogeneity of devices derived from the continuous technological changes demand the explicit modeling of all kinds of variability, which must be managed with efficient mechanisms, suitable even for ultra-large systems. However, MASs lack mechanisms to effectively deal with the different degrees of variability present in these kinds of systems. *Software Product Line* (SPL) technologies have been successfully applied to different domains to explicitly model variability in hardware or variations

in both system requirements or user preferences [5]. In order to overcome the limitations of traditional MAS development processes, the MAS-PL initiative proposed combining SPL and MAS to promote reuse and variability modeling in the context of complex systems [6].

SPLs principally rely on variability models (e.g., feature models), which are used to specify the common assets shared by a set of products of the same family and the variable elements that will or will not be in the final product configuration, only if they are selected. On the other hand, MASs foster system development in terms of system objectives, plans and quality goals that should be fulfilled. Integrating goal models and SPLs brings several advantages to the CPS development processes. Goal models improve communication with stakeholders because they can better express their expectations about, for example, what technology they want. It is easier for a stakeholder to express that they want a system to help him with the power management of a smart home than selecting the configuration variants of the heater, blinds and home appliances that consume less energy. In addition, all kinds of requirements, including non-functional requirements or soft goals are explicitly identified and modeled using any goal model alternative. Therefore, SPL can greatly benefit from Goal-Oriented Requirements Engineering (GORE) approaches that complement variability models with a stakeholder viewpoint. With goal models it is easier to represent intentional variability, which facilitates the exploration of design alternatives [7].

However, combining GORE and SPL approaches [7,8] to improve CPS development is not enough. CPS construction also needs processes that provide support to systematically manage the evolution of SPLs according to changes in goal models (e.g., new security goal for a smart home system) and technology (e.g., new home appliance). Indeed, current market trends are volatile, imposing novel technologies and requirements that must be incorporated into running systems quickly. Therefore, we need processes that automatically propagate changes (e.g., substitute an obsolete technology) to agents running in multiple devices, considering that each of them has a different and customized configuration. In this paper, we combine goal-driven and SPL approaches to drive the evolution of MASs embedded in IoT devices that take into account the environmental context in the decision-making to generate plans that fulfill CPS goals.

In [9] we have defined an SPL development process that performs a model driven generation of agents (*Self-StartMAS* [10]) embedded in heterogeneous smart objects with different degrees of self-management objectives. With this approach, instead of generating a single system, we can generate custom-made MASs, composed of agents embedded in smart devices, with capacity for self-management. In this paper, we enhance this process with a Goal-Oriented perspective which is integrated with a *model driven evolution process*. We name the joint work of these processes GOSPEL, a Goal-Driven Product Line approach for evolving MAS in the IoT. GOSPEL uses goal models specified in iStar 2.0 [11] to describe system requirements and goals, and CVL (Common Variability Language) [12] to specify variability. It also defines a set of algorithms that manage the evolution of iStar 2.0 and CVL models synchronously, and propagate these changes to the running system, assuring CPS consistency at all times. We present and validate our approach for a home energy management system. This home energy assistant system will be evolved to self-adapt its decision making, or the home appliances it manages, to new requirements. As some CPSs are capable of managing hundreds of computing devices, we validate our approach to test whether or not it is applicable to ultra-large CPSs. Indeed, our approach becomes especially valuable when the devices to propagate the changes number in the hundreds, since we automate this process, while ensuring the consistency of the resulting system.

The remaining sections of this paper are organized as follows: Section 2 presents a brief overview of the models used in this paper (iStar 2.0 and CVL) and Self-StartMAS. Section 3 presents and discusses a revision of existing work, which explores the combination of GORE and SPL, and the evolution based on goals and requirements. Section 4 presents the *GreenManager* system, a case study used to illustrate our proposal, and the challenges that motivate our approach. Section 5 describes GOSPEL, the Goal Oriented SPL process for IoT agents. Section 6 details our process for the evolution of MAS due to new emerging goals and requirements. Section 7 validates our approach while Section 8 discusses the threats to validity. Finally, Section 9 concludes the paper.

2. Background

In this section we present an overview of the models used in the work presented here and Self-StarMas (the target agent technology of our processes) in a nutshell. We use CVL to model variability and iStar 2.0 to model goals.

2.1. The Common Variability Language

Variability of SPLs can be specified using different modeling languages. Although feature models are very popular in the SPL community, the Common Variability Language (CVL) [13] was recently proposed as a richer alternative. Both variability languages can be used in our proposal, but here we have opted to use CVL. CVL is a domain-independent language for specifying and resolving variability over any instance of any language defined using an MOF-based metamodel (e.g., UML) which is called the *base model*.

In a CVL process (see Fig. 1(a)) all the variations of the *base model* are specified in the *variability model*. The *variability model* (see top of Fig. 1(b)) is a specification of the base model variabilities and their relationships, and it is defined in two steps. Firstly, the *variation points (VP)* (i.e., what varies) are marked over the *base model* (see center of Fig. 1(b)). There are different types of variation points: to indicate the *existence* of an element; to *substitute* a particular part of the base model; to perform a *value assignment* of a particular slot of the model.

Secondly, to complete the *variability model* we need to define the *Variability Specifications* (VSpec) separately from the base model. These entities are specifications of abstract variability that are organized in tree structures (VSpec trees, where nodes are called VSpec) (see Fig. 1(b)). Additionally, it is possible to specify explicit constraints also known as *cross-tree constraints*. The VSpecs can be divided into three kinds: *choices*, *variables* and *classifiers*. *Choices* represent yes/no decisions or features that can appear in the base model or not. *Variables* are VSpecs that require a value to be resolved. *VClassifiers* (or clonable features) mean having to create instances or clones and then providing per-instance resolution for the VSpec in its sub-tree. In the models shown in following sections we use *choices* and variability classifiers (*VClassifiers*). *Group multiplicities* are used to apply restrictions over the number of children of a VSpec that can be chosen. Furthermore, the appearance of VSpecs in a selection can be optional or mandatory.

The effect of the variability model on the base model is specified by *binding variation points*, which link 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…*), and this selection makes up the *resolution model*. So, the VSpec tree with the features (or VSpecs) selected is called *resolution*
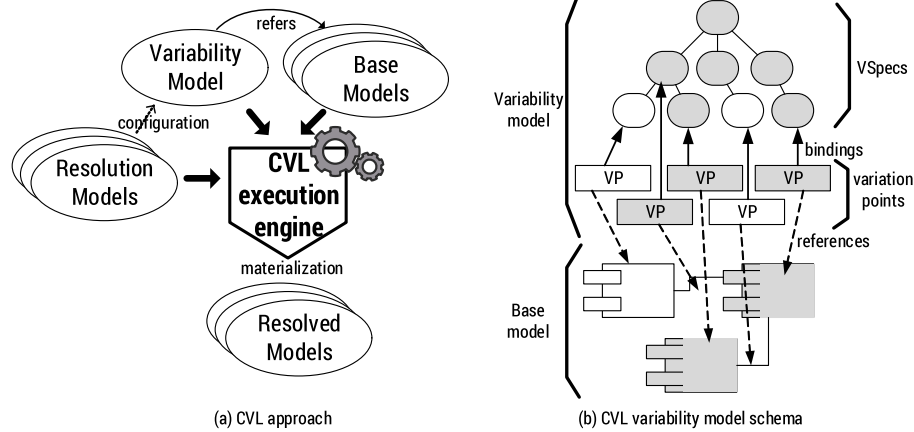
(a) CVL approach

(b) CVL variability model schema

Fig. 1. Common Variability Language overview as specified in [12].

model (i.e., a specific configuration of the variability model). This *resolution model* holds dependencies entailed by the VSpec tree structure and the cross-tree constraints of the *variability model*.

When one of the *resolution models* is selected, the CVL execution engine (see Fig. 1(a)) is executed and produces the *resolved model*, a product model, fully described in the MOF-based meta-model, which is a variation of the *base model* according to the choices that have been made in the *resolution model*.

2.2. The iStar 2.0 core language

The iStar 2.0 core language [11] is an evolution of the early requirements language i* (iStar). The main purpose of this language is to facilitate the stakeholders' understanding of the problem solution. With this goal, the language takes into consideration the social and the intentional dimensions of the problem. This language is a goal-oriented and actor-oriented modeling language. Therefore, iStar 2.0 models describe the goals of the system to be developed, and how different actors cooperate to accomplish them.

The notation of the language considers social entities (actors, agents and roles), intentional elements (goals, qualities, tasks and resources), social dependencies and links between intentional elements. The language has two perspectives for system modeling: (i) the strategic rationale, which shows all the intentional elements associated with their actors and their dependencies; and (ii) the strategic dependency, which only shows the social part of the model.

The iStar 2.0 core language has a tool[2] that works with the full specification of the language. Fig. 2 shows a minimal example with some of the main elements of the language. The actor *Researcher* wants *i* models created*. To do so he/she executes the task *Use piStar* that helps to create *Good quality* models. Additionally, if he/she cooperates with the agent *piStar*, which participates in the role *Modeling Tool*, he/she can *Export image* of his/her models in PNG of SVG. We will provide more details about the semantic of the iStar 2.0 language later in Section 5.1, when describing its use in our SPL process.

2.3. Self-StarMAs agents

Self-StarMAs is an agent system, which adapts and extends standard agent technologies to help in the development of IoT applications. In this system, we can distinguish two parts: Self-StartMAS [10], a set of self-adaptive cooperating agents developed for lightweight devices; and the agent platform where the agents are deployed. Self-StartMAS agents can be executed on top of different agent platforms like Jade [14] by just selecting the appropriate plug-in. Concretely, in the work presented here, we deploy Self-StartMAS agents in Sol [15], an agent platform suitable for agents in the IoT.

Self-StarMAS agents can be embedded in typical devices of the IoT like sensor motes, handheld personal devices or personal computers. A distinguishing feature of these agents is that they have self-adaptation capabilities, adapted to the resources of the devices where agents are embedded [3]. For instance, agents for handheld devices support goal-oriented self-adaptation, while agents embedded in sensor motes support reactive adaptation based on rules. The main features of the Sol agent platform are the support for communication of agents in heterogeneous devices, coping with heterogeneous transport protocols (WiFi, Bluetooth and ZigBee) and the support for group communication, which are often required by pervasive systems.

The joint work of Self-StarMAs and Sol provides the necessary means for developing IoT applications and in particular Home Automation Systems. Self-StarMAs agents can take advantage of the Sol agent platform, to communicate through different transport protocols and send multicast messages to a group of related agents. With this approach, the functionality of the IoT applications is decomposed into a set of Self-StarMAS cooperating agents that use the Sol agent platform for location and communication between agents.

3. Related work

This section presents a revision of existing proposals that explore the combination of goal models and SPL, and also those which propose evolution processes based on goals. Finally, and in relation to our case study, we mention previous work that applies agent technology to smart homes and home automation systems.

This distinction reflects that, to the best of our knowledge, there are no proposals that encompass MAS, SPL, goal modeling and intentional evolution in the same work.

3.1. Goal-oriented software product lines

Bridging the gap between goal models and SPLs is not a new topic, and there are different approaches that have combined goal models into SPL [7,8,16–20]. The main reason of these proposals is to show how the features of an individual product fulfill the
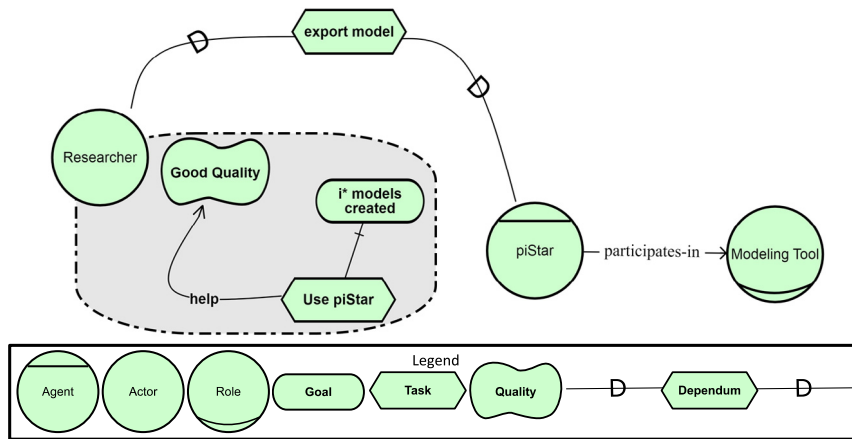

Fig. 2. Example of iStar 2.0 Goal Model in the piStar tool.

stakeholders' goals and, therefore, how goals can be used to drive and improve feature selection in the feature model configuration process. Existing proposals show that combining goal and feature models can be done through two different approaches: including goal models in the life cycle of the SPL [7,8,16,21]; and using goal models to generate feature models [17–20].

In the work presented in [7], a goal model, referred to as the *family goal model*, is integrated in the SPL life cycle to illustrate variability in the intentional space. The family goal model is described using an extension of the Goal Requirements Language (GRL) [21], and represents intentions in product lines and the difference of intentions for various products. The family goal model and the feature model are connected by bidirectional mappings, which facilitates traceability between stakeholders' objectives and features.

The approach by Noorian et al. [8] proposes the use of goal models for the derivation of products taking into account stakeholder requirements and preferences. The goal model and the feature model are linked using the Quality-Centric Feature Modeling Method, which makes it possible to set quantitative relations between intentional elements and features. Following a staged configuration process, the featured model is gradually configured to select an optimal set of features that can maximize stakeholders' needs. The authors propose a formalization of the structure of the feature model, the stakeholders' objectives and their selections as an Integer Linear Programming problem, which drives a semi-automated staged configuration process.

Another interesting contribution is the work presented in [16], which proposes to use the Aspect Oriented User Requirement Notation (AoURN) [22] at the requirement analysis phase of SPL life cycle. AoURN is a modeling and reasoning framework that combines goal-oriented, scenario-based, and aspect-oriented techniques for requirements engineering. This approach focuses on the analysis of the interaction between stakeholders' requirements and feature models, and considers the use of tools that help to identify, prioritize and choose products, taking into account stakeholder requirements. As part of this proposal, the authors propose a mapping between AoURN and feature models by the definition of a URN profile that models common concepts of a feature model in a URN profile as a specialized AoURN goal model.

In current proposals that integrate goal-models into the SPL life cycle (at the requirements phase), the goal model and the variability model are developed in parallel and linked later. This lack of automation in the mapping makes the application of these approaches difficult when the models contain a lot of elements. Additionally, the aforementioned approaches provide different mappings for the manual connection of models that focus on just one type of element of the goal model. For example, [7] considers a mapping from tasks to features, while in [16] the mapping is performed between goals/soft-goals and features. Here we can find an important difference to our work, which limits the use of these proposals as part of our approach. Note that, with these mappings, some of the information provided by the goal model gets lost in the feature model (e.g., goals in [7,8] and tasks in [16]). In our work, the use of MAS-PL, which combines MAS and SPL, allows all the information provided by the goal model to be taken into account, and this is included in the feature model use to derive the MAS, and also to automate the generation of models.

Another distinction between our approach and those aforementioned [7,8,16] is their reliability on non-standard goal models and extensions. Both approaches use extensions of GRL/URN for goal modeling. The use of non-standard models reduce the set of available tools, which complicates their automation. Our approach uses iStar 2.0 standard models, which have helped to automate the generation of models and mappings.

As mentioned, another approach to combine GORE and SPL is to support the generation of feature models from goal models. This approach is intended to support traceability, and/or complement and enhance the semantics and justification of the features. Yu et al. in [19] explored the relations between goal models and feature models to approach a requirement traceability problem. They provided a model-driven tool to support a systematic process for generating a feature model from a goal-oriented model. This directly supports configuring feature models via preferences as stated by end-users over higher level soft goals. Both models can be further edited iteratively using the OpenOME tool. The derivation of products is supported by the information provided by the soft-goals that represent user preferences. A limitation in the work of Yu et al. [19] is that it requires the variation points to be manually defined in the goal model in order to be able to generate the feature model. In contrast, our work automatically derives the information of the variation points from the structure of the iStar model.

IStarLPS [17] proposes adapting the SPL process to include iStar to enhance and help to define feature models, providing an early identification of variability. The concept of cardinality is added to iStar to represent variability in the iStar models. The domain engineering process of the IStarLPS approach deals with the development of the iStar goal model, which is extended with cardinality, and the feature model. The approach provides a set of hypotheses to produce the feature model from iStar models.

The use of aspectual goal models is also considered in [18] and [20]. The work in [18] defines an aspect-oriented requirement language (PL-AOVGraph) that integrates feature models

with requirements engineering concepts, and proposes a bidirectional mapping that allows to generate feature models from PL-AOVGraph models and viceversa. Silva et al. [20] propose using aspectual iStar to capture common and variable features in SPL requirements. The generation of the feature models is supported by a set of heuristics.

Finally, we wish to remark that none of the aforementioned approaches consider the generation of the application architectures for deployment. Those publications that consider the derivation of products like [17–20] do not support the connection between configurations of the feature models and the final system. In our approach, CVL is used to connect variability models with architectures by means of binding variation points, and supports the generation of the application architectures using the CVL engine.

3.2. Goal-oriented software evolution

Even though software evolution is usually related to changes in stakeholders' needs and requirements, the research in goal-oriented or intentional evolution is not profuse [23]. Nonetheless, there are some approaches that deal with the evolution of requirements or goals and how this evolution impacts on the system, including the context of SPLs [24–26] and MAS [27].

The work in [28] proposes a methodology for relating business process models (modeled using BPMN) to high-level stakeholder goals (modeled using KAOS) to be used in dynamic environments where organizational goals and/or processes are constantly emerging.

The work in [23] is focused on requirements that cause the evolution of other requirements, and how they affect the system at runtime. Although this does not explicitly consider SPLs, it introduces the concepts of variation points and control variables to model variability at the requirements level. This approach deals with anticipated and explicitly specified requirements changes, and their impact on the system can be carefully predicted.

The work in [25] supports the evolution of the SPL to cope with changes in the stakeholders' requirements. The EvoPL framework supports an integrated model-driven approach to plan the evolution of SPLs in the production of different releases of a product at feature level. Evolution is considered a sequence of versions of a feature model. The proposed integrated planning process starts modeling goals and requirements using goal models. A set of incremental model transformations allow the automated generation of a plan to generate the evolution of the feature model from existing versions. Goals are refined into concrete requirements and the features to fulfill them, but it is not explained how this refinement (i.e., goal-feature relationship) is established. The result is a release plan which has to be realized by an implementation, but which does not consider how to address the propagation of changes at the architectural level.

The work in [26] focuses on analyzing evolving requirements and contexts to evaluate the potential variations in features. The authors propose a problem-oriented and value-based analysis method for this variability evolution analysis. The method confirms that tracing back to requirements and contextual changes is an effective way to understand the evolution of variability in SPLs.

In the MAS domain, evolution is the ability of a group of agents to cooperate and evolve so as to adaptively search an optimization design space guided by a strategy [29] or an evolutionary algorithm [30]. The fact that strategies and evolutionary algorithms, embodied in agents, evolve over time is an extension of evolutionary computation [31] in itself. The work of Gross and Yu [27] is closer to our definition of evolution, as it proposes combining goal models and agents for software evolution. This approach focuses on modeling the relationships between business goals and system qualities, and how these goals are met during architectural design. In particular, modeling must encompass changes to business goals over time and their effects upon a system's architecture. Goals serve as a guide in the search for design alternatives, and as criteria for choosing between them. This approach uses the agent concept to model human organizations as well as technical components.

In conclusion, these approaches [23–27] consider the evolution of the goal models but they do not propagate the changes to the variability model nor the final architectural elements. In fact, no work takes into account both mapping between goal models and variability models and the subsequent automatic updating of that mapping when the requirements evolve.

3.3. Agent technology for home automation systems

Agent technology and Smart House and Home Automation systems are linked in several proposals [32–35]. The role of agents is to act as an intelligent distributed control system [34], decision support [36], or in diagnostics and monitoring [33]. With these capabilities, the home can adapt the control of many aspects of the environment such as climate or water consumption, lighting, maintenance, and multimedia entertainment. Intelligent automation of these activities can reduce the amount of interaction required from the inhabitants, reduce energy consumption and other potential wastage, and provide a mechanism for ensuring the health and safety of the dwellers [33,37]. In home automation systems focused on power management the goal of agents is to adapt the power consumption to the power resources available, and vice-versa [38–40], usually taking into account the inhabitants' comfort criteria [32,37].

As we stated in Section 2.3, the implementation of GreenManager is based on Self-StarMAS agents [9]. This agent technology has been used for the implementation of home automation systems [3,41] and offers great support to manage the heterogeneity of these environments. Current home automation products support a limited set of devices and communication protocols, but our proposal can incorporate diverse hardware and communication technologies even after product deployment. Finally, our agents have self-adaptation capabilities that facilitate the development of systems with multiple and contradictory concerns like power management and user comfort. Then, Self-StarMAS appropriately addresses the main challenges of smart homes [42] related to the adaptability to everyday domestic contexts and the flexibility to modify adaptation policies on the fly.

4. Developing agent-based CPS applications

In this section we present the *GreenManager* system, a case study used to illustrate our proposal. We also discuss the challenges that our approach has to address in order to improve the development of agent-based IoT applications, using goal models and SPLs.

4.1. Case study: The GreenManager system

Residential buildings represent the most important part of the energy consumption of highly-populated areas, which worsens the greenhouse effect. This problem is being mitigated thanks to the popularization of smart home systems [42], which are making home appliances smarter and include sustainable modes for power and water consumption saving. Another feature of these kinds of home appliances is the possibility to program a specific task to be done at some point in the future (e.g., the laundry can be delayed until night time), thereby avoiding the

peak periods of power consumption, which is when the energy price is highest. In addition, energy tariff models imposed by energy providers are now more complex and dynamic than in the past [32,38], making the task of minimizing the energy cost very complex to do manually. All this poses new challenges in the automation of home energy management. Inhabitants need some kind of assistant tool, an energy manager, to help them in making complex decisions with different goals, depending on their current situation.

The *GreenManager* system is an agent-based home automation system, whose goal is to manage the smart home power. Sustainability goals at home encourage inhabitants to use resources more effectively. Experience has shown that small changes in habits (e.g., turning off lights, selecting an appropriate home appliance program) can result in a substantial energy saving. The goal of the system is to help occupants optimize home power consumption without imposing undue technological complexity, effort or inconvenience. *GreenManager* manages home appliance operation to adapt the power consumption to the available power resources and taking into account user comfort criteria, and certain power constraints. Therefore, the energy management system has to reach a compromise between the priorities of the user in terms of usability and in terms of cost considering providers' tariffs, while satisfying technological constraints of home appliances, in-house devices and infrastructures.

The *GreenManager* system has to interact with the home appliances of the user to automate the starting and stopping of service tasks, and the monitoring and control of resource consuming activities. Monitoring helps to detect faults or abnormal operating situations. For instance, during peak consumption periods and when the energy price is high, it would be possible to postpone some consumption activities until late night, when energy price and demand are usually lower. The best candidates to delay the start of the service are those silent home appliances whose service is not a priority to users, like washing machines, for example. To model this situation, we consider two types of home appliances, the silent ones and those whose work cannot be delayed like the oven or the vacuum cleaner. The *GreenManager* plans these activities taking into account the information provided by the electric power company. However, if many energy consuming tasks are programmed when the energy price is cheaper, it is necessary to control the demanded power so that it does not exceed the power limit contracted, or the power line limit. Automatic power management is more challenging and useful when the availability and price of the energy varies. It is extremely complicated for users to manage the distribution of energy consumption in a dynamic pricing context. Therefore, the system has to reach a compromise between the inhabitants' comfort and the energy costs (consumption, price, or both) while satisfying the technological and green conditions of home appliances and devices.

The system behavior relies on a Multi-Agent paradigm. Each agent is embedded inside a home appliance or piece of equipment (computing unit). These devices can affect the environment (e.g., thermal-air) or automate a task (e.g., washing). The agents in the devices can cooperate with others to find acceptable power consumption while maintaining comfort. Fig. 3 details part of the MAS architecture integrated into the system to solve the power management problem. The MAS comprises several types of agents embedded in each home appliance: (*WashingMachineAgent*, *DryerAgent*, *DishwasherAgent*, *CooktopAgent*, *ExtractorHoodAgent* and *OvenAgent*). It also has the agent that manages its functioning, the *GreenManager*, which considers the electricity cost and the limitations of the electric power of the house.

In the normal functioning of the MAS, the agents embedded in home appliances whose programs and start time can be delayed (i.e., washing machine, dryer or dishwasher) request a time slot to start and finish an activity from the *GreenManager*. In order to assign the best time slot, the *GreenManager* uses information provided by the other agents of the house and their demand on power consumption. Each agent is aware of, in general terms, the user preferences for the home appliance use and start time (e.g., as soon as possible or at a certain time). So, in their decisions there is a trade-off between cost, electrical power limitations and meeting the user's preferences. On the other hand, there are home appliances whose start time cannot be postponed like the oven or the cooktop. Once the agents embedded in the oven and the cooktop (the *OvenAgent* and the *CooktopAgent*) inform the *GreenManager* that the user has requested their service, and taking into account the current state of the other home appliances and the electrical power limit of the house, this agent (the system manager) can take three actions: (i) to do nothing because the start of the cooktop or the oven is not going to exceed the electrical power limit of the house; (ii) to ask the corresponding agent to use a low consumption program (if it is available); or (iii) to suspend or delay the planned start of one of the home appliances. The action (ii) is a recommendation that the user can accept or cancel, in the case that the user does not accept it, *GreenManager* takes action (iii).

4.2. Challenges for developing agent-based CPS applications

In this section we discuss the challenges of developing and managing the evolution of CPS including IoT devices such as the *GreenManager*.

One of the first challenges is **C1: managing the heterogeneity in hardware and software**. Since devices (home appliances and personal devices) are heterogeneous in nature, the software (the agents in the *GreenManager*) embedded in them is also heterogeneous. This means that each agent will have its own internal architecture, with goals and plans adapted to the specific capacity of the device. However, the goals, the agent platform and other shared characteristics have to be consistent for each agent comprising the MAS. So, we need to model agents that have to share some features, but at the same time the agent configuration must be customized to each device.

Another challenge is **C2: the customization required of the system when deployed for different customers**. This challenge is clearly exemplified in the *GreenManager*. Developers should work on a new version of the application for each house in which the *GreenManager* has to be deployed. Each house has a different set of home appliances or energy tariffs and in addition, final users will have different preferences. The use of techniques for systematic reuse of common assets and customization of non-common variants could greatly facilitate this work.

Another important challenge is **C3: management of the evolution of the system**. The original MAS deployed in a given home, which is initially adapted and generated for specific home appliances, personal devices, and currently applicable goals or energy tariff models, must evolve to meet changes. These new requirements could be demanded by the final stakeholders or as a result of technological upgrades or even new rules or policies. During a system's operational life new devices may appear, while others must be replaced, the tariff models can change, and in general, the running *GreenManager* system configuration has to be adapted to new requirements. This means that when a new requirement appears, we need to calculate the changes that must be made to each agent. Considering that for any given system this may entail many agents, the manual calculation of changes would be a costly and error-prone task, it would therefore be better to make these calculations automatically.

Once the changes have been calculated in the design models, we need to automatically and coherently propagate these changes
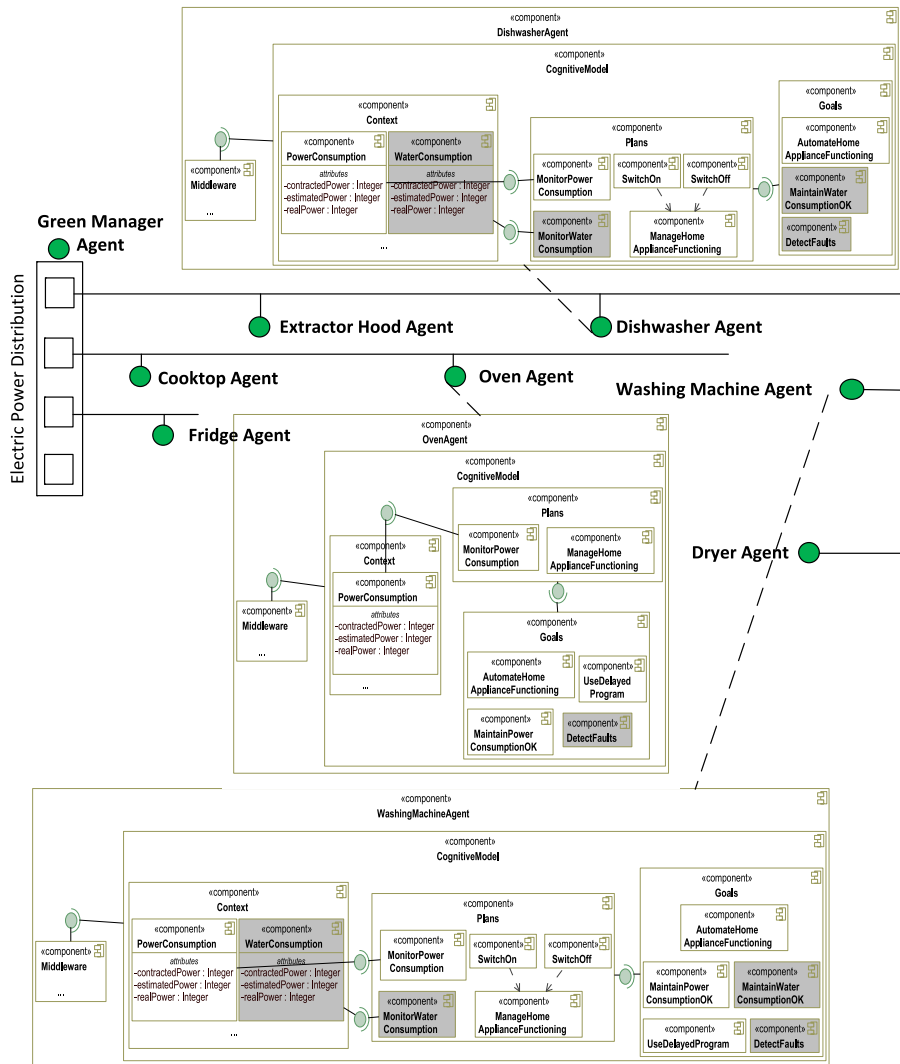
Fig. 3. Architecture of the *GreenManager* system.

to all the running agents. We need to ensure that agents will continue interacting and cooperating in a coherent way, but according to the new requirements. Some changes will entail the modification of a single agent and others may imply modifying all or several agents. For instance, in our case study, to manage the water consumption together with the power consumption, new goals and plans need to be added, but only in those agents embedded in home appliances that consume water. Equally, a new system requirement that affects all the agents is the detection of faults in the functioning of the home appliances. These changes imply having to add, remove or modify multiple components in the architecture of the agents (see components marked in gray in Fig. 3).

The objective of the *GreenManager* is to automate some user decisions with respect to the power distribution between the home appliances, with the goal of contributing to the planet's sustainability. So, another challenge here is **C4: to include goals specific to the energy efficiency of home appliances in system modeling**. This should be done in a language that can be understood by all the application stakeholders.

The development and evolution processes that we propose in the following sections try to address all these challenges. With this goal, we rely on techniques that facilitate the management of the variability of the system, the systematic re-use and the

Table 1
Challenges addressed in the related work.

Challenges	Works	Comment
C1	[7,8,16–20,24–26]	Define SPL to manage heterogeneity.
C2	[7,8,16–20]	Use goal models and SPL to support system customization taking into account user requirements.
C3	[23–31]	Manage requirement evolution at the SPL level.
C4	[32–35,37–40]	Consider goals of energy efficiency.

evolution of the IoT system. In Section 3, we present different works that address some of the challenges presented in this section (see Table 1). However, as far as we know, there is not a single approach that deals with all the challenges raised in this section.

5. GOSPEL: A Goal-Oriented SPL process for IoT agents

In order to address the challenges related to the development of CPS and IoT systems like the *GreenManager*, we propose a Goal-Oriented SPL process based on the iStar 2.0 and the CVL modeling languages (see Fig. 4).
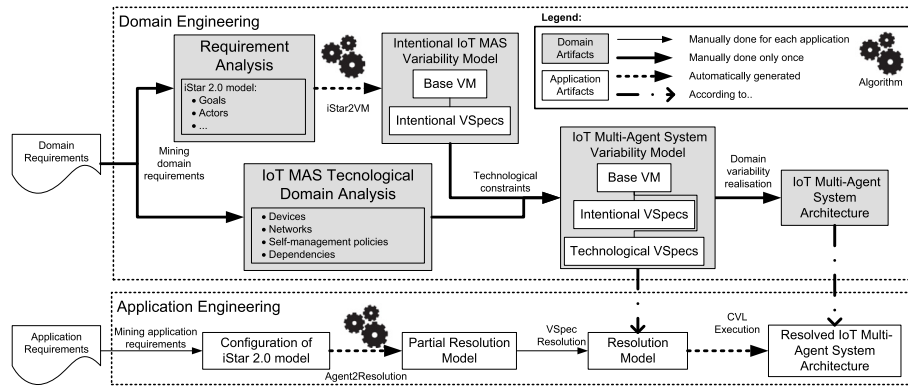
Fig. 4. GOSPEL, goal-oriented and SPL process to develop MAS for the IoT.

CPS and IoT applications like Home Automation Systems, could greatly benefit from an SPL approach [43,44]. SPL approaches are composed of the Domain Engineering (DomE) and the Application Engineering (AppE). In the DomE, the process has to produce the common models and the core assets of the product family. The AppE entails the creation and configuration of one of the family members. CPS and IoT Systems are composed of a heterogeneous set of devices and network technologies [45]. So, the identification and management of this variability is performed at the DomE level. Additionally, these systems should be customized for each of the environments in which they will be deployed. This task is performed at the AppE level, using the processes for the derivation of new family members.

The GOSPEL process presented in this section is implemented in iStar 2.0 (i.e., iStar) and CVL. We have selected iStar because of its maturity and popularity in the Early Requirements stage. Furthermore, it is tool supported and it has been used in other approaches that integrate goal models with SPL [17,19]. In the case of CVL, we use this language because it is a variability modeling language that supports all the characteristics of the most popular feature modeling mechanisms (i.e., VClassifier, variables, propositional cross-tree constraints, groups, ...). In addition, this language allows feature models (i.e. VSpec trees in CVL) to be connected with the architecture of the system, which facilitates the generation of products from the SPL (see Section 2.1).

5.1. Goal-oriented domain engineering

The DomE phase entails the definition of several software artifacts that will be reused in each product of the family, so it only has to be performed once (see Fig. 4). Our DomE phase, starts with the elicitation, representation and management of stakeholders' requirements for the family of applications. These tasks are supported by iStar and its tool. The result is a goal model, which contains a goal-oriented modeling of the family of systems. The goal model of the *GreenManager* (see Fig. 5) includes the main actors of our case study, the *GreenManager*, the agents that manage the home appliances, i.e., *HomeApplianceController*, and other supporting actors like the *Electric Company* that provides information about the energy tariff and the limit of the contracted power. Due to limitations of space, we focus on the description of the *GreenManager*.

The main goal of the *GreenManager* system is the management of the power consumption of a smart home (see *Power consumption Managed* at the top of Fig. 5). This general goal is refined into the sub-goals *Operation Schedule Generated* and *Operation Schedule Executed* by means of an *AND-refinement* iStar operator. This kind of refinement represents the sub-goals as part of the parent goal. The purpose of these sub-goals is to support the work planning of the different home appliances. To achieve these goals we need additional goals like *Appliance Data Collected*, *Schedule Obtained* and *Schedule Strategy Selected*. This last goal is *OR-refined* by the goals *Adapt to user preferences*, *Adapt to user needs*, *Adapt to Energy tariff*, *Adapted to power limit* and *Trade-off comfort efficiency*. So any of these sub-goals can be achieved for fulfilling the *Schedule Strategy Selected*. Goals can also be refined in tasks using *AND-refinement* (e.g., *Collect Appliance Status*) or *OR-refinement* (e.g., *Schedule Operation Remotely*). The meaning of the AND-refinement is that the sub-task must be fulfilled to accomplish the goal, while in the OR-refinement, the sub-task is a particular way for fulfilling the parent goal. The *GreenManager* system also considers some qualities that will guide the search for ways to achieve goals like *Sustainable Home* or *Avoid Power Interruption*. Goals, tasks and other non-functional requirements impact on these qualities by means of contribution links. At this point, we have different situations because this impact could be high (e.g., *Energy Efficiency* strongly contributes to the *Sustainable Home*) or low (e.g., *Adapt to user needs* goal helps achieve *user comfort*). Additionally, it is possible to model relationships with a negative impact by means of the *hurt* and *break* contributions, but they have not been considered in our case study.

In order to accomplish its goals or to obtain resources, the *GreenManager* cooperates with other actors. These scenarios are modeled using *dependencies*. For instance, to accomplish the task *Schedule Operation Remotely*, *GreenManager* has to accomplish *Appliance Controlled*, which requires the accomplishment of the goal *Appliance Smart Control* of *Appliance Basic Controller role*. *Roles* in iStar are used to characterize an abstract class of an actor. Therefore, we have used them to model the controllers of home appliances. These controllers have a similar behavior in each home appliance, but in an iStar model for the *GreenManager* family it does not make sense to define specific features that will vary from one product to another. For example, it is not possible to know the number of controllers (i.e., the home appliances controlled by an agent) we need for a specific house, or some additional specific behavior that certain home appliances might provide, until we need to generate a specific *GreenManager* product at the application engineering phase. Therefore, using iStar roles, we can support the modeling of the *structural variability*, which is essential to model CPS and IoT systems.

The next step in the DomE process is the automatic generation of the *Intentional Variability model* of our SPL using the information of the iStar model. To do so, we have implemented the *iStar2VM* algorithm (see Algorithm 1), which makes a mapping between the iStar and the CVL model. This algorithm generates the VSpec tree that is adapted to the *base variability model (Base VM)* (Fig. 4) defined by Self-StarMAS. The Base VM specifies a MAS in terms of an agent platform and the set of agents,
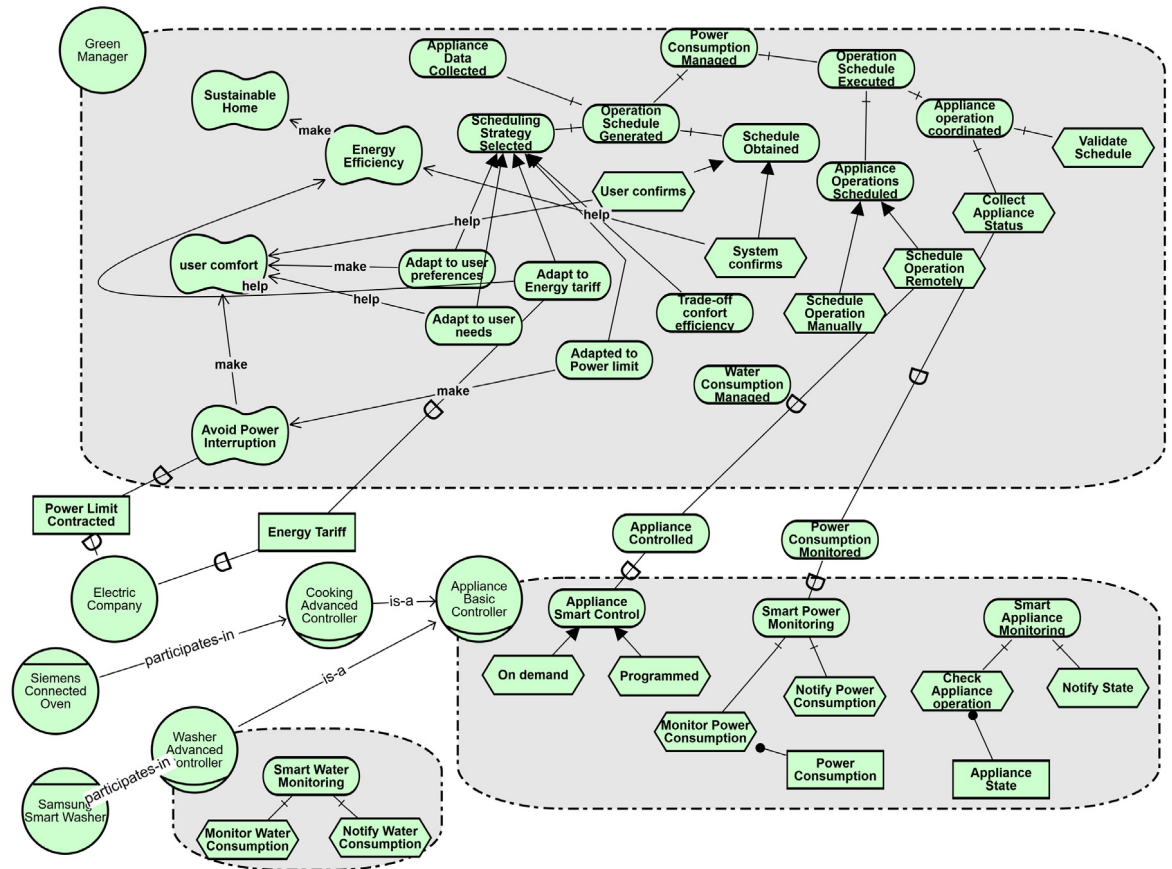
Fig. 5. Goal model in iStar 2.0 of the *GreenManager* MAS.

where each agent contains the device features in which it is embedded and a Cognitive model, expressed in terms of *Context*, *Plans* and *Qualities* (see Fig. 6). One of the advantages of using CVL is that it allows cardinality to be assigned to the features of the VSpec tree, which in CVL language is known as VSpec trees with VClassifiers (see Section 2.1). In these kinds of models, VSpecs can be qualified with a cardinality (e.g., [1..*]) meaning that a VSpec (or feature) can be cloned with the entire subtree rooted in that VSpec. The advantage of clonable VSpecs is that each clone can have a different configuration of the variability, e.g., to use different goals or plans, adapted to each clone's characteristics. In our approach the cardinality represents the number of agents, so each agent can have a different configuration depending on the device in which it is embedded. As shown in Fig. 6 for the *GreenManager*, we can explicitly model the structural variability of the agents for home appliances (e.g., the sub-trees under `WaterAdvancedController[1..*]`, `CookingAdvancedController[1..*]` and `ApplianceBasic-Controller[1..*]`) with the agent functionality customized to the characteristics of each home appliance.

So, Algorithm 1 adds the intentional variability modeled in iStar to the base model where agents are clonable features. The iStar models have three sources that can be interpreted as intentional variability: (i) the *OR-refinement* links, (ii) the contribution links for qualities, and, (iii) the roles, which stand for the structural variability of the system, meaning that we can have many instances of a role in the system (e.g., the home appliance controller). The algorithm creates a VSpec tree which contains all the types of agent roles (line 5, Algorithm 1), actors (line 8), goals and tasks (line 18). To generate these agents, the algorithm follows the mapping shown in Table 2. Each time an element is added to the VSpec, its relationships with other elements are added by

means of parent–child relationships or cross-tree constraints. For example, in the case of an *OR-refinement* between two elements of the same type (line 28), the result is an XOR group in the VSpec tree, meaning that only one of the alternatives can be selected. At the same time, a resource needed by a task generates a cross-tree constraint (line 44). In the case of qualities (line 49), they are modeled as optional VSpecs and if they are related with links *Make* or *Help*, the algorithm generates a cross-tree constraint with the feature that it helps or makes.

Returning to Fig. 4, at this point of our GOSPEL process, we have a VSpec tree which contains the intentional variability of the *GreenManager*. This model provides a view of the intentional elements of the MAS and how they are realized using different features, however, it cannot be used to generate a running version of the *GreenManager*. We need information about the technologies that can be used to implement this system. Gathering this information is the goal of the task *IoT MAS Technological Domain Analysis*, which is accomplished by domain experts. For our case study (see Fig. 6), we have considered the technologies of Self-StarMAS and the Sol agent platform (see Section 2.3 and Fig. 6). Therefore, the work of the *GreenManager* is supported by two types of devices that have sensors related to battery consumption and network technology coverage. Additionally, these applications use different network technologies, such as WiFi or ZigBee, and require self-management capabilities for their correct functioning. Therefore, we can find many variation points common to IoT devices, or related to the connectivity required by the specific the *GreenManager* that we are developing.

As explained in Section 2.1, the CVL variability model includes: (1) an abstract part (the tree structure at the top of Fig. 6), and (2) a realization part (middle of Fig. 6) with the *variation points* that connects VSpecs with the software architecture of the agent
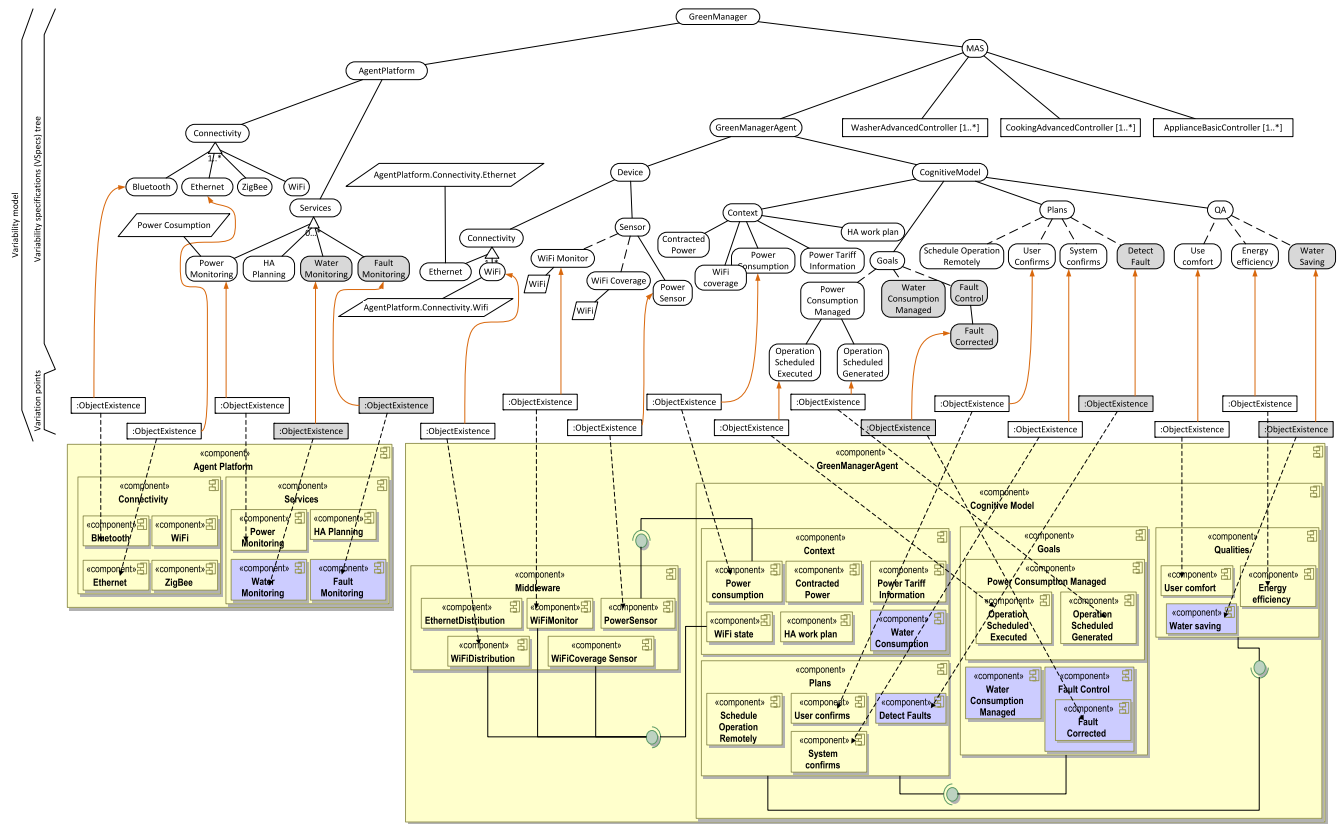
Fig. 6. Partial view of the variability model of the Green Manager Multi-Agent System.

iStar 2.0	Variability model
Actor	Mandatory feature
Role	Clonable feature [1..*]
Agent	Instance of the clonable feature
Primary goals	Mandatory feature
Tasks	Optional feature
OR refinement same type (goal to goal or task to task)	Alternative feature (XOR group)
AND refinement same type (goal to goal or task to task)	Mandatory feature
OR refinement different type (goal to task or task to goal)	Cross-tree constraint (XOR implications)
AND refinement different type (goal to task or task to goal)	Cross-tree constraint (AND implications)
Resources	Optional feature
Qualities	Optional feature
Resource needed by task	Cross-tree constraint (implication)
Task contributes to quality	Cross-tree constraint (implication)
Goal dependency	Mandatory feature
Task dependency	Optional feature
Resource dependency	Optional feature
Quality dependency	Optional feature

(bottom of Fig. 6). For example, the *Operation scheduled generated* VSpec (derived from the task with the same name in iStar that postpones the execution of a home appliance service) is linked to a software component that implements this functionality inside the agent. In CVL this is called *the variability realization domain*, meaning that the variation points are linked with the software architecture of the product family.

5.2. Goal-oriented application engineering

As we stated in the introduction to this section, one of the challenges in the development of IoT applications, like *GreenManager*, is the complexity of the variability models that have to be understood by final stakeholders. We address this challenge with a two-step configuration process of the VSpec tree. In the first step (*Configuration of iStar 2.0 model* in Fig. 4), final stakeholders resolve the intentional variability of the system, and in the second step (*VSpec resolution*), software architects and system developers resolve the technological variability of the system, so the output is the *Resolution Model*.

The *Configuration of the iStar 2.0 model* is supported by the concept of agent in the iStar language. While actors and roles are abstract entities, agents represent concrete, physical manifestations, such as persons, organizations or software. In order to resolve the intentional variability of actors and roles, we model agents that are versions of these entities without variability. Fig. 5 shows two agents that are derived from the role *Home Appliance Controller*, one for the washing machine and another for cooking appliances.

This iStar model with agents is used as the input of the *Agent2Resolution* algorithm (see Algorithm 2) to generate a partial configuration of the variability model of our IoT system. For each agent in the iStar model, *Agent2Resolution* resolves its corresponding intentional elements in the VSpec tree (see lines 14–16). If the agent participates in a *Role*, then the elements that correspond to the agents in the VSpec tree are under a VClassifier (i.e., a clonable feature). Therefore, *Agent2Resolution* creates an instance of this agent in the resolution model (line 6) with the intentional elements attached to the agent (lines 7–11).

Algorithm 1 Algorithm for the automatic generation of variability models

INPUT: *istar* ▷ i* (istar) model
OUTPUT: *VT* ▷ VSpec Tree

```
1:  VT ← VSpecTree()                                    ▷ New empty VSpec Tree
2:  root ← VT.addVSpec('GreenManager', 'mandatory', None)  ▷ The root has no
    parent
3:  addStructuralVariability(VT)          ▷ This information is not in the i* model
4:  Actors, Roles ∈ istar
5:  for all rol ∈ Roles do                                    ▷ Mapping for roles
6:      VT.addVClassifier(rol, '[1..*]', root)
7:  end for
8:  for all act ∈ Actors do                                  ▷ Mapping for actors
9:      VT.addVSpec(act, 'optional', root)
10: end for
11: for all act ∈ Actors ∪ Roles do
12:     Deps ← Dependencies(act)
13:     Goals, Tasks, Resources, Qualities ∈ IntentionalElements(act) ∪ Deps
14:     GoalsF ← VT.addVSpec('Goals', 'optional', VT.vspec(act))
15:     TasksF ← VT.addVSpec('Tasks', 'optional', VT.vspec(act))
16:     ResourcesF ← VT.addVSpec('Resources', 'optional', VT.vspec(act))
17:     Qualities ← VT.addVSpec('Qualities', 'optional', VT.vspec(act))
18:     for all gt ∈ Goals ∪ Tasks do           ▷ Mapping for goals and tasks
19:         if ¬∃VT.vspec(gt) then
20:             if gt ∈ Goals then
21:                 VT.addVSpec(gt, 'mandatory', GoalsF)
22:             else
23:                 VT.addVSpec(gt, 'optional', TasksF)
24:             end if
25:         end if
26:         for all child ∈ Goals ∪ Tasks | gt.refinesTo(child) do
27:             if gt.type() = child.type() then
28:                 if gt.refines().type() = 'OR' then   ▷ OR refinement child of the
    same type
29:                     VT.addVSpec(child, 'alternative', gt)
30:                 else              ▷ AND refinement child of the same type
31:                     VT.addVSpec(child, 'mandatory', gt)
32:                 end if
33:             else
34:                 diff_children ∈ {c ∈ Goals ∪ Tasks | gt.refinesTo(c) ∧ gt.type() ≠
    c.type()}
35:                 if gt.refines().type() = 'OR' then   ▷ OR refinement children of
    different type
36:                     VT.addConstraint(VT.vspec(gt) ⟹ ⨁ VT.vspec(c))
                                                       c∈diff_children
37:                 else        ▷ AND refinement children of different type
38:                     VT.addConstraint(VT.vspec(gt) ⟹ ⋀ VT.vspec(c))
                                                       c∈diff_children
39:                 end if
40:             end if
41:         end for
42:     end for
43:     for all r ∈ Resources do                          ▷ Mapping for resources
44:         VT.addVSpec(r, 'optional', ResourcesF)
45:         if ∃t ∈ Tasks | r.needBy(t) then
46:             VT.addConstraint(VT.vspec(t) ⟹ VT.vspec(r))
47:         end if
48:     end for
49:     for all q ∈ Qualities do                          ▷ Mapping for qualities
50:         VT.addVSpec(q, 'optional', QualitiesF)
51:         if ∃t ∈ Tasks | t.contributesTo(q) then
52:             if t.contributesTo(q).type() ∈ {'Make','Help'} then
53:                 VT.addConstraint(VT.vspec(t) ⟹ VT.vspec(q))
54:             else
55:                 VT.addConstraint(VT.vspec(t) ⟹ ¬VT.vspec(q))
56:             end if
57:         end if
58:     end for
59: end for
60: Return VT
```

Algorithm 2 Algorithm for the generation of partial configuration models

INPUT: *istar, VT* ▷ i* (istar) model, feature model
OUTPUT: *VTresolution* ▷ partial resolution of the VSpec tree

```
1:  VTresolution ← ResolutionModel()              ▷ New empty resolution model
2:  Agents, Actors, Roles ∈ istar
3:  instance ← 1
4:  for all ag ∈ Agents do
5:      if ∃rol ∈ Roles | ag.participatesIn(rol) then   ▷ Instance of VClassifier
6:          VTresolution.createInstance(VT.vspec(rol), instance)
7:          for all ie ∈ IntentionalElements(rol) do  ▷ Goals, Tasks, Resources, and
    Qualities
8:              if ie ∈ IntentionalElements(ag) then
9:                  VTresolution.select(VT.vspec(ie), instance)
10:             end if
11:         end for
12:         instance ← instance + 1
13:     else if ∃act ∈ Actors | ag.participatesIn(act) then  ▷ Selection of feature (not
    clonable)
14:         for all ie ∈ IntentionalElements(act) do  ▷ Goals, Tasks, Resources, and
    Qualities
15:             if ie ∈ IntentionalElements(ag) then
16:                 VTresolution.select(VT.vspec(ie))
17:             end if
18:         end for
19:     end if
20: end for
21: Return VTresolution
```

CVL engine, the architecture configuration *Resolved IoT Multi-agent system architecture* is obtained (shown in Fig. 3), with each agent configured to fulfill their specific needs.

6. Evolving the multi-agent system in GOSPEL

Once a specific architecture configuration customized to each agent has been generated and deployed, the MAS evolves due to different factors. These factors include the emergence and disappearance of technologies, something common in IoT environments, where the market demands new functionalities or the adoption of new laws. Furthermore, once the changes issued are defined, they should be propagated to the running instances of the system. Finally, in the case of ultra-large systems the modifications must be performed in hundreds of agents. All this greatly complicates the manual evolution of IoT applications.

In a previous contribution, we proposed an evolution process for MAS-PL in IoT environments [46]. The process focused on the propagation of changes performed in the variability model and the architecture of the SPL to the deployed agents. However, performing system changes in the variability model and the architecture is adequate for stakeholders with a deep knowledge of the visible and no visible features of the system, like developers or software architects. Even for these stakeholders, the evolution process could be challenging when the size of the models is considerable and the relationships between features and requirements of the system become highly complex. Additionally, as we explain in Section 4.2, these models are not suitable for stakeholders without a technical background, like final users. In order to overcome these limitations, we propose performing the evolution process beginning with the goal models.

Our evolution process (see Fig. 7) is similar to the one presented in [46], which is based on the algorithms presented in [47]. Moreover, we add two additional steps and three algorithms that focus on iStar models. The output of the process is a weaving model that updates the agents deployed in the MAS.

We illustrate the work of this process using our case study. Let us suppose that two new requirements are incorporated: (1) controlling and managing the water consumption, apart from the power consumption, in those home appliances that consume
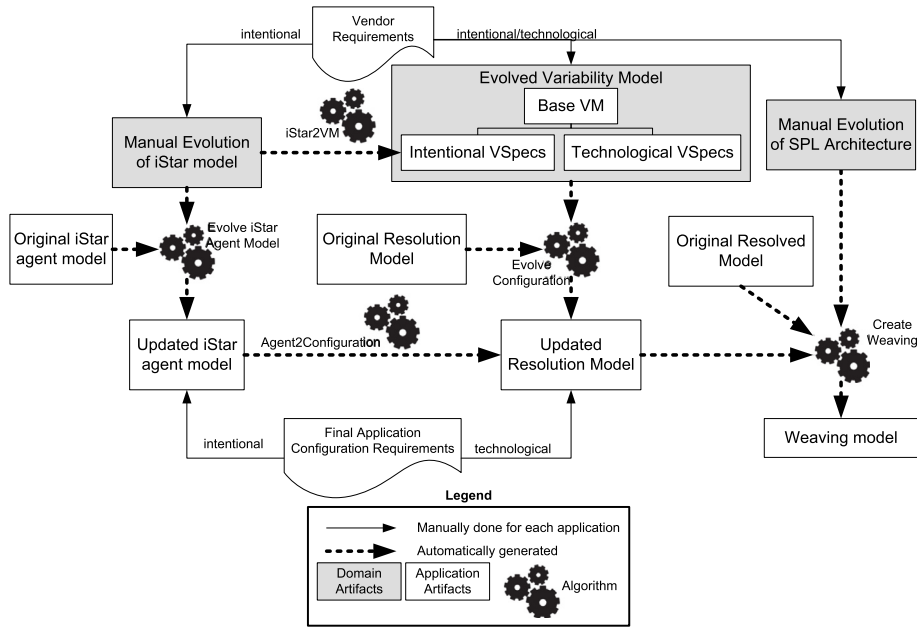
The *Agent2Resolution* algorithm (in Algorithm 2) only resolves the intentional variability (i.e., goals, qualities, plans and intentions) and the structural variability of our system, (i.e., the number of agents deployed in the *GreenManager*). Finally, software architects and developers select the appropriate features under the *GreenManager* and the other agents, and after executing the

Fig. 7. Evolution process of MAS-PL agents.

water, and (2) detecting errors and malfunctions in all the home appliances.

We divide the description of our algorithm into three parts: (i) evolution of the intentional models; (ii) evolution of the variability model; and (iii) evolution of the agents' architecture. In the usual scenario, the three parts of the process are sequentially executed. However, it is possible that the evolution process only affects the variability models. This may occur when the evolution is only related to technological requirements of the system. For example, imagine that the *GreenManager* has to support NFC, this will not affect its goal model because it does not include elements to model specific network technologies. However, it will affect the variability model and the architecture of the SPL, and these changes must be propagated to the deployed agents of the *GreenManager*.

6.1. Intentional evolution

The first step in the evolution process is the **manual evolution of the iStar model**. For example, we can add additional actors or intentional elements, or we can modify the relationships between them. For our case study, the iStar model will be modified as shown in Fig. 8 (new and modified elements are in yellow). We have added new goals to support the control and management of water consumption (*Water Consumption Managed*) and to support the detection of faults (*Fault Control*). These goals are supported by new sub-goals like *Fault Corrected* and new sub-tasks like *Notify Fault*. In order to achieve these goals, the *GreenManager* cooperates by means of new dependencies with *Washer Advanced Controller* and *Appliance Basic Controller*.

The second step is to **update the iStar 2.0 agent models** according to the modifications made in the iStar model. As we have stated in Section 5.2, agents represent specific instances of actor/role iStar elements that have a physical representation in the real world. Therefore, agents do not contain variability and they must conform to the corresponding actor/role element. In order to propagate these changes, we use the algorithm *Evolve iStar Agent Model*, which uses the original and the updated iStar agent models as input (see Algorithm 3). The algorithm analyzes the differences between actors/ roles from the updated version

and the original agents. Firstly, it focus on the intentional elements that must be removed (lines 7–9), and then it focuses on the elements that must be added to the agents (lines 10–23). When an intentional element is added or removed, then its relationships are also removed. The result of the execution of this algorithm can be seen in Fig. 6, modifications made by Algorithm 3 are highlighted in gray. The *GreenManager* agent has been updated with the goals *Water Comsumption Managed*, *Fault Control* and *Fault Corrected*. In addition, this agent has new plans and qualities to manage such as *Detect Fault* and *Water Saving*.

Once the iStar agent model has been updated, the evolution process may require the participation of the final user of the application. This happens when the iStar model has been updated with optional elements (i.e., agents, intentional elements or dependencies). Then, the final user of the application must decide whether he/she wants these optional elements in *GreenManager* deployed in his/her home. The process cannot continue until the final user has made a decision on this. When the final user decisions are included in the iStar agent model, then the next step of the process is **to update the resolution model** of the deployed system. This task is performed automatically using the algorithm *Agent2Configuration* (see Algorithm 2), which has been described in Section 5.2.

The last step of the intentional evolution is **to evolve the variability model** taking into account the modifications introduced in the iStar model. This task is accomplished automatically using the algorithm *iStar2VM* (see Algorithm 1) presented in Section 5.1.

6.2. Software product line evolution

The evolution process of the SPL is based on the algorithms presented in [47], which were intended for general SPL architectures and show good results in terms of efficiency and complexity.

The first step in this part of the process is to **manually evolve both the variability model and the base software architecture** of the MAS according to the technological upgrades. This step results in a set of changes in the resolution model.

The second step is to **propagate the changes made in the variability model** consistently in the resolution model of all the existing agents. This step is done at the abstract level (i.e., at the
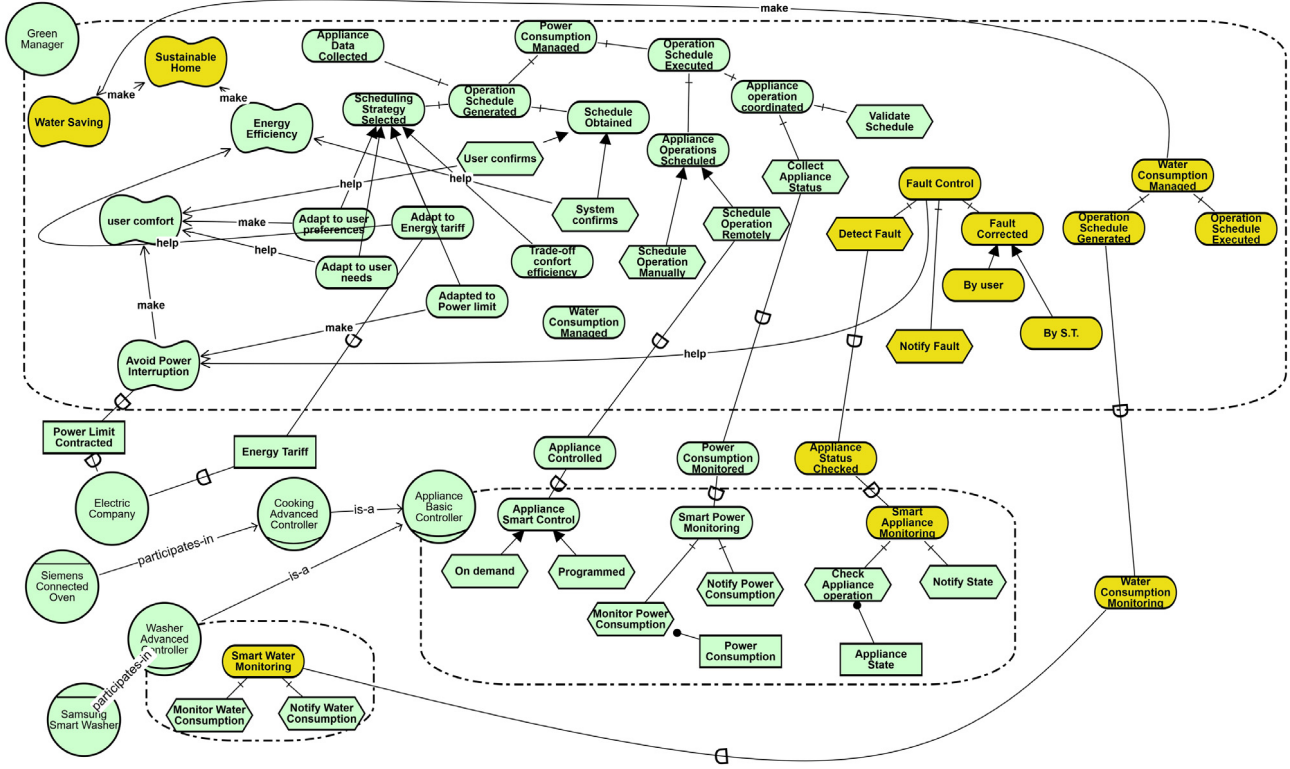
Fig. 8. Evolved goal model of the GreenManager in iStar.

Algorithm 3 Algorithm to update an agent model from an evolved i* model.

INPUT: *istarM*, *agentsM*   ▷ evolved Actor/Role model, previous Agent model
OUTPUT: *evolvedM*   ▷ evolved Agent model

1: *evolvedM* ← *copy*(*agentsM*)
2: *Actors* ∈ *istarM*
3: *Roles* ∈ *istarM*
4: *Agents* ∈ *agentsM*
5: **for all** *act* ∈ *Actors* ∪ *Roles* **do**
6:   **for all** *ag* ∈ *Agents* | *ag.participatesIn*(*act*) **do**
       ▷ Delete intentional elements that are not in the evolved model anymore
7:     **for all** *ie* ∈ *IntentionalElements*(*ag*) | *ie* ∉ *IntentionalElements*(*act*), ∀*act* ∈ *Actors* ∪ *Roles* **do**
8:       *evolvedM.delete*(*ie*)   ▷ Delete intentional elements and its associations
9:     **end for**
         ▷ Add new intentional elements (including dependencies)
10:     *Deps* ∈ *Dependencies*(*act*)
11:     **for all** *ie* ∈ *IntentionalElements*(*act*) ∪ *Deps* | *ie* ∉ *IntentionalElements*(*ag*) **do**
12:       **if** *ie.refines*().*type*() = 'AND' **then**   ▷ mandatory intentional element
13:         *evolvedM.add*(*ie*)
14:       **end if**
15:     **end for**
                 ▷ Add associations
16:     **for all** *ie* ∈ *IntentionalElements*(*act*) | *ie* ∈ *IntentionalElements*(*ag*) **do**
                 ▷ Add new contributions
17:       **if** ∃*q* ∈ *Qualities*(*act*) | *ie.contributesTo*(*q*) ∧ *ie.contributesTo*(*q*) ∉ *evolvedM* **then**
18:         *evolvedM.add*(*ie.contributesTo*(*q*))
19:       **end if**
                 ▷ Add new neededBy associations
20:       **if** ∃*r* ∈ *Resources*(*act*) | *ie.neededBy*(*r*) ∧ *ie.neededBy*(*r*) ∉ *evolvedM* **then**
21:         *evolvedM.add*(*ie.neededBy*(*r*))
22:       **end if**
23:     **end for**
24:   **end for**
25: **end for**
26: **Return** *evolvedM*

---

resolution model) to make all the changes explicit and ensure the consistency of the changes with all existing and new constraints. To do so, we use the **Evolve Configuration** algorithm [46] that generates a new resolution model from the previous resolution model and from the evolved variability model, respecting the new constraints.

Finally, the last step consists in **automatically propagating the evolution changes** calculated in the previous steps to the previously **deployed software architecture**. This is done through the use of model-to-model transformation (in the case of a models@runtime approach) [48], changing the configuration files or scripts of the agents [49], or directly changing the deployed code [50]. To do that, we use the **Create Weaving Model** algorithm, which generates a model in charge of propagating the changes to the architecture of the agents when CVL is executed.

6.3. Evolution management in CVL

In order to apply these steps, we have identified the actions over the elements of the variability model that need to be carried out as a consequence of an evolution scenario in the MAS: (1) adding or removing VSpecs that correspond to adding or removing a new goal, plan, context, etc.; (2) adding, removing or modifying group multiplicity that VSpecs may have; (3) modifying instance multiplicity that VClassifiers have; (4) adding or removing OCL constraints between VSpecs; and (5) modifying the variability of a VSpec (i.e. a mandatory VSpec is made optional or vice versa). Note that the modification of a VSpec can be defined by removing a previous VSpec and adding a new one. The same is true of OCL constraints modification. Composite VSpecs can be seen as a collection of VSpecs under a VSpec. Variation points in the CVL model are only affected when a VSpec is added or removed.

Taking these modifications to the variability model into account, Fig. 6 shows the VSpec tree of the MAS with some of the evolution changes presented in our example in Section 4.1 (new features shown in gray in Fig. 6). That is, the VSpec tree is modified by adding new choices to control the water consumption and to detect faults in the home appliances' functioning. Fig. 6 shows three new goals, `Water Consumption Managed`, *Fault Control* and `Fault Corrected`; a new plan to control faults, `Detect Faults`; and a new quality `Water saving`. Additionally, services of the `Agent Platform` have been updated with `Water Monitoring` and `Fault Monitoring`.

After evolving the variability model, the changes have to be propagated in all the previous agent's configurations taking into account the new requirements of the system.

7. Evaluation

In this section, we evaluate the efficiency of the algorithms of our SPL and evolution processes taking into account their complexity and performance in terms of execution time.

7.1. Complexity of the algorithms

The computational efficiency of an algorithm is the number of basic operations it performs depending on its input length [51]. To evaluate the efficiency of the mapping and evolution algorithms, we analyze the time complexity of them. In our case the input length (n) is the size of the iStar model that corresponds with the number of elements: actors, roles, and intentional elements (goals, tasks, resources, and qualities). Evolution algorithms also take into account the number of agents (a) in the iStar model as the number of instances of the clonable features according to the defined mapping. Let us consider the following basic operations of iStar models and feature models: (i) comparison between elements, where elements can be actors and intentional elements for the iStar model; and features and selections for the variability model and its resolutions; (ii) adding a new element to a model; and (iii) deleting an element from a model.

Table 3 shows the complexity in terms of the Big $\mathcal{O}$ notation for our algorithms. Here we focus on the algorithms related to the intentional variability represented in the iStar models. The algorithms related to the technological variability were previously evaluated in [46].

The *iStar2VM* algorithm (Algorithm 1) has cubic time complexity ($\mathcal{O}(n^3)$) in n, while the *Agent2Configuration* (Algorithm 2) and *Evolve iStar Agent Model* (Algorithm 3) algorithms have also cubic time complexity ($\mathcal{O}(an^3)$), but taking into account n (the size of the model) and a (number of agents in the model). Specifically, for the *iStar2VM* algorithm, mapping the actors or roles to features takes $\mathcal{O}(n)$ operations (lines 5–10), but for each actor/role, mapping their goals and tasks takes $\mathcal{O}(2n^3 + 12n^2)$ operations (lines 18–42 in Algorithm 1). Mapping resources (lines 43–48) and qualities (lines 49–58) takes $\mathcal{O}(8n^2)$ and $\mathcal{O}(10n^2)$ operations, respectively. In the *Agent2Configuration* algorithm, mapping agents to instances of clonable features takes $\mathcal{O}(2an^3 + 3n)$ operations (lines 4–12 in Algorithm 2), and mapping agents to instances of simple actors (i.e., not clonable features) takes $\mathcal{O}(5n)$ operations (lines 13–19). Finally, in the *Evolve iStar Agent Model* algorithm, updating all the agents by deleting and/or adding new intentional elements together with their dependencies and associations, takes $\mathcal{O}(32an^3)$ operations (lines 6–24 in Algorithm 3).

Algorithms that manage the intentional variability are more complex than those managing technological variability [46]. This is because algorithms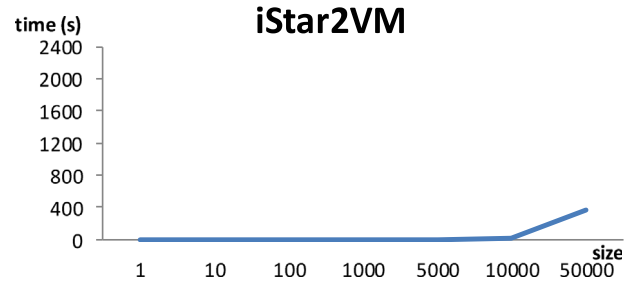 that evolve intentional variability have to deal with two kinds of models (i.e., the iStar model and the variability model), while the algorithms to manage the technological variability only consider one model (i.e., the variability model). In fact, in the approach presented in this paper, the complexity of the algorithms that evolve the variability model is the same but with a bigger input (n) because, after the mapping procedure, the variability model contains both the intentional variability and the technological variability.

7.2. Performance of the algorithms

To evaluate the performance of the algorithms we have generated random iStar models and iStar agent models with a variable number of elements (from 1 up to 50,000), with the following proportions: 2.5% of actors, 5% of roles, 7.5% of agents, 32.5% of goals, 40% of tasks, 10% of qualities, and 2.5% of resources. So, we have executed our algorithms measuring their execution time. The experiments were performed on a desktop computer with Intel Core i7-4770, 3.40 GHz, 16 GB of memory, Windows 10
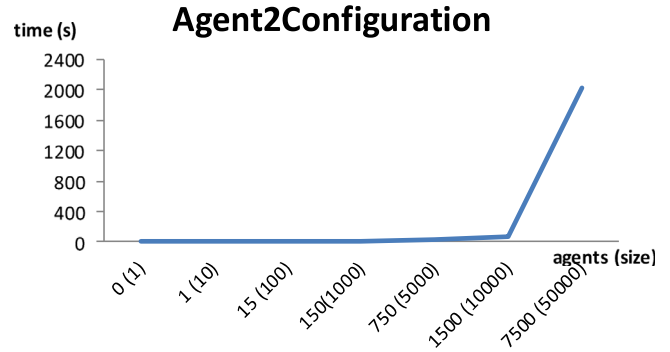

Fig. 9. Performance of the iStar2VM algorithm.

Fig. 10. Performance of the Agent2Configuration algorithm.
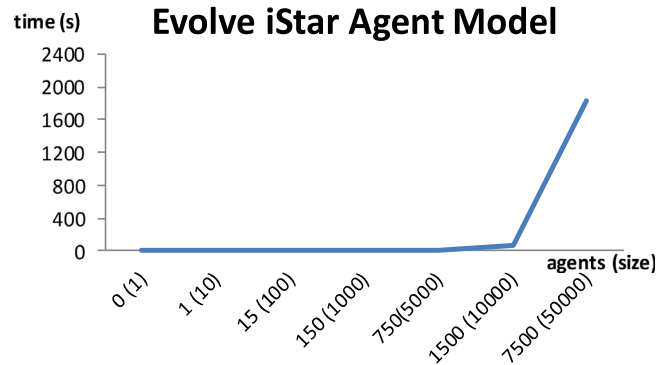
Fig. 11. Performance of the Evolve iStar Agent Model algorithm.

Table 3
Complexity of the algorithms.

	Algorithm	Computational complexity
Intentional variability	iStar2VM (Algorithm 1)	$\mathcal{O}(4n^3 + 44n^2 + 10n + 2)$
	Agent2Configuration (Algorithm 2)	$\mathcal{O}(4an^3 + 5n + 1)$
	Evolve iStar Agent Model (Algorithm 3)	$\mathcal{O}(32an^3 + n)$
Technological variability	Evolve Configuration [46]	$\mathcal{O}(5n^2)$
	Difference Configuration [46]	$\mathcal{O}(4n^2)$
	Create Weaving Model [46]	$\mathcal{O}(3n^3)$

64 bits and Python 3.5.2. All algorithms and scripts have been implemented in Python and are available online together with the results of this evaluation,[3] so they can be reused and to allow readers to replicate these experiments.

We have performed 30 runs for each model size (from 1 up to 5000) and 10 runs for huge models (from 10,000 up to 50,000), and we have calculated the averages, standard deviations, and medians of the execution times (in seconds).

Figs. 9, 10, and 11 show the results of the empirical experiments for the three algorithms presented in this paper. The efficiency depends on the iStar model size (n) for the mapping algorithm and in the size (n) and the number of agents (a) for the evolution algorithms. For instance, to perform a mapping from an iStar model of 5000 elements to a feature model, the iStar2VM algorithm (Algorithm 1) takes around 4 s (Fig. 9). In a fictitious model of 50,000 elements (impossible to build manually), the algorithm takes around 6 min to generate a valid and consistent feature model. Algorithm 2, it takes even longer to generate the resolution model with the different configurations of the agents (Fig. 10). For instance, it takes 75 s to generate a resolution model from an iStar agent model of 10,000 elements with 1500 agents, but for an iStar agent model of 50,000 elements with 7500 different agents the algorithm takes 33 min. Updating those configurations of the agents (Algorithm 3) demonstrates a similar efficiency as shown in Fig. 11. For instance, to evolve an iStar agent model of 10,000 elements with 1500 agents, Algorithm 3 takes 69 s, while for the huge model of 50,000 elements with 7500 agents it can take around 30 min. Therefore, we can conclude that the efficiency is acceptable for huge models (e.g., models with a 50,000 elements and 7500 agents). Models of this size are very difficult to find in the real world in the context of Home Automation Systems.

The other algorithms responsible for evolving a resolution model and the deployed configurations of the agents were analyzed in detail in [46]. The results, shown in Figs. 12, 13, and 14, prove that managing variability models is more efficient than managing iStar models. This is owing to the fact that in variability models we only have to deal with one concept (i.e., VSpecs), while in iStar model we have to deal with many kinds of concepts such as intentional elements (goals, tasks, qualities, resources) and also with actors, roles, and agents, apart from also dealing with VSpecs of the variability model.

We think that our scalability and performance results are quite good since it is possible to manage huge iStar models in seconds. We would like to compare these results with similar works. However, the works that generate feature models using goal models like [17–20] do not evaluate the proposed algorithms [18,19], nor even provide the code making it impossible to compare them with our work [17,20]. In the case of the evolve configuration algorithm it is possible to compare it with a similar work [52] that generates a configuration from a feature model (instead of a CVL model as we do), and that calculates the differences
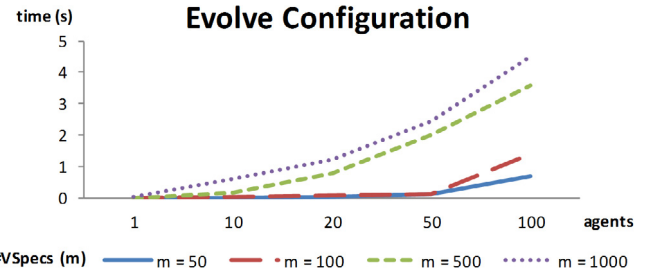

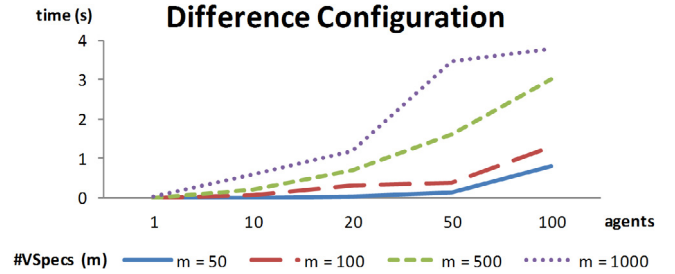**Fig. 12.** Evaluation of the evolve configuration algorithm.

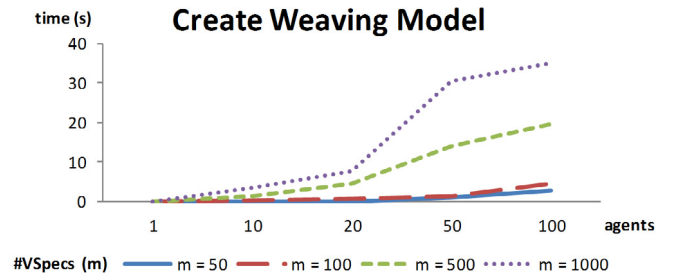**Fig. 13.** Evaluation of the difference configuration algorithm.

Fig. 14. Evaluation of the create weaving algorithm.

between configurations as we do. The algorithms of Gamez and Fuentes take around 120 s to create a configuration with a total of 11,000 elements (500 agents with 22 elements each), and around 60 s to obtain the differences between configurations, while our algorithms take, for the same models size, only 75 s to create the configuration and 4 s to obtain the differences. In fact, our algorithms have been evaluated for models with 7500 agents and a total of 50,000 elements.

8. Threats to validity

This section discusses the threats to validity with respect to the four groups of common threats to validity: internal validity, external validity, construct validity, and conclusion validity [53].

8.1. Internal validity

The internal validity examines whether or not the experimental results are influenced by other factors apart from those considered in the experiments.

Complexity of the technological variability. The complexity evaluation of our iStar2VM mapping algorithm (Table 3) considers only intentional variability expressed in the iStar model. However, the resulting variability model also contains the structural variability of the MAS-PL that is generated from scratch from specific templates (line 3 in Algorithm 1), since that information is not in the iStar model. The complexity of generating the structural variability was previously evaluated with the algorithms that manage the variability models [47]. In the worse case scenario, complexity is quadratic in the size of the model, it being smaller than the complexity of dealing with iStar models as discussed in Section 7. Therefore, the complexity of adding the structural variability does not affect the global complexity of our iStar2VM mapping algorithm.

8.2. External validity

The external validity analyzes whether the results obtained in the experimentation can be generalized or not.

Generalization of the results. We have assumed that the structure of the iStar models specify the intentional elements of the actors and roles, while the structure of the iStar Agent models specifies only the agents as instances (configurations) of the roles in the iStar models. While the iStar2VM mapping algorithm (Algorithm 1) can be applied regardless of the structure of the iStar models, the evolution algorithms: Agent2Configuration (Algorithm 2) and Evolve iStar Agent Model (Algorithm 3) must respect the structure of Self-StarMAS agents defined as instances of roles.

Scalability of the proposal. We have automatically generated iStar models of up to 50,000 elements, and 7500 agents to evaluate our proposal. From 10,000 elements, the performance of our algorithms considerably decreases (from seconds to minutes). However, an iStar model of that size is not realistic in practice. In fact, the main iStar model of the MAS-PL needs to be manually defined by the stakeholders during domain engineering, as it is impossible to automatically define that model.

8.3. Construct validity

The construct validity analyzes the completeness and consistency of the evolved models.

Completeness and consistency. Our evolution algorithms only validate that the generated models are built conforming the variability metamodel, but they do not check for additional properties such as dead features, redundancy constraints, or false-optional features. To guarantee the consistency of our models, our algorithms include some scripts to generate the variability models in different formats such as those supported by the S.P.L.O.T. [54] and FeatureIDE [55] tools that allow us to analyze the completeness and consistency of the models.

8.4. Conclusion validity

The conclusion validity relates to the reliability of the experiments and whether they can be replicated with the same results.

The threats related to the reliability of the experiments are represented by the mapping performed by the algorithms and the models are taken as input. As can be derived from the evaluation section (in Section 7) and the subsequent discussion, since the mapping and models (both goal and VSpec tree model) rely on the concepts and entities of metamodels, if the metamodels do not change, the results of the experimentation would be the same. Additionally, the algorithms were tested with input goal models and variability models, generated randomly. In all cases, the algorithms worked fine and generated well-formed iStar and variability models, i.e., models that are correct instances of iStar and CVL metamodels respectively [56].

9. Conclusions

This paper has described GOSPEL, a model-driven evolution process that uses goal models specified in iStar 2.0 to describe system requirements and goals, and the CVL language to specify variability. The process makes it possible for stakeholders who are not familiar with variability models to define new requirements and goals in iStar, which is a GORE language. It also defines a set of algorithms that manage the evolution of iStar 2.0 and CVL models synchronously, and propagate these changes to the running system, assuring CPS consistency at all times. The approach has been presented and validated for a home energy management system. This home energy assistant system has been evolved to self-adapt its decision making, or the home appliances it manages, to new requirements. As some CPSs are able to manage hundreds of computing devices, we have validated our approach to test whether or not it is applicable to ultra-large CPSs. Indeed, validation shows that our approach becomes especially valuable when the number of devices that have to propagate the changes number in the hundreds, since we have automated this process, thereby ensuring the consistency of the resulting system.

The GOSPEL process continues and enhances the work in [46], which presented the *GreenManager* system, providing an extensible approach that allows addressing the goal-driven configuration and evolution of MAS by means of SPL for heterogeneous domains and platforms.

Despite the fact that in this contribution the focus is on a home energy management system, GOSPEL is generic enough to deal with different MASs in the context of IoT frameworks or platforms. Therefore, our approach can be generic enough to change the configuration needs, tailored to incorporate heterogeneous application domains in the context of the IoT. We are currently working on the application of GOSPEL to a case study of Electric companies.

As future work, we plan to extend our work with a method to assist stakeholders in the configuration of products using softgoals. In this sense, we are analyzing how to introduce methods that quantify the relationships among elements in the goal model and features like the Quality Centric Feature Modeling Method [8].

Acknowledgments

References

[1] C.G. García, D. Meana-Llorián, J.M.C. Lovelle, et al., A review about smart objects, sensors, and actuators, Int. J. Interact. Multimedia Artif. Intell. 4 (3) (2017) 7–10.

[2] J.C. Preciado, Á.E. Prieto, R. Benitez, R. Rodríguez-Echeverría, J.M. Conejero, A high-frequency data-driven machine learning approach for demand forecasting in smart cities, Sci. Program. 2019 (2019).

[3] I. Ayala, M. Amor, L. Fuentes, Self-configuring agents for ambient assisted living applications, Pers. Ubiquitous Comput. 17 (6) (2013) 1159–1169, http://dx.doi.org/10.1007/s00779-012-0555-9.

[4] M.S. Taboun, R.W. Brennan, An embedded multi-agent systems based industrial wireless sensor network, Sensors 17 (9) (2017) http://dx.doi.org/10.3390/s17092112, URL https://www.mdpi.com/1424-8220/17/9/2112.

[5] K. Pohl, G. Böckle, F.J.v.d. Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer-Verlag, New York, Inc., 2005.

[6] I. Nunes, C.J.P. de Lucena, U. Kulesza, C. Nunes, On the development of multi-agent systems product lines: A domain engineering process, in: M.-P. Gleizes, J.J. Gomez-Sanz (Eds.), Agent-Oriented Software Engineering X, Springer, Berlin, Heidelberg, 2011, pp. 125–139.

[7] M. Asadi, G. Gröner, B. Mohabbati, D. Gašević, Goal-oriented modeling and verification of feature-oriented product lines, Softw. Syst. Model. 15 (1) (2016) 257–279, http://dx.doi.org/10.1007/s10270-014-0402-8.

[8] M. Noorian, E. Bagheri, W. Du, Toward automated quality-centric product line configuration using intentional variability, J. Softw. Evol. Process 29 (9) (2017) 1–26, http://dx.doi.org/10.1002/smr.1870.

[9] I. Ayala, M. Amor, L. Fuentes, J.M. Troya, A software product line process to develop agents for the IoT, Sensors 15 (7) (2015) 15640–15660.

[10] I. Ayala, J.M. Horcas, M. Amor, L. Fuentes, Using models at runtime to adapt self-managed agents for the IoT, in: M. Klusch, R. Unland, O. Shehory, A. Pokahr, S. Ahrndt (Eds.), Multiagent System Technologies, Springer International Publishing, Cham, 2016, pp. 155–173, Best Paper Award MATES 2016.

[11] F. Dalpiaz, X. Franch, J. Horkoff, iStar 2.0 language guide, 2016, https://arxiv.org/abs/1605.07767. (Accessed 31 March 2019).

[12] CVL Submission Team, Common Variability Language (CVL), OMG revised submission, 2012, URL http://www.omgwiki.org/variability/.

[13] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G.K. Olsen, A. Svendsen, Adding standardized variability to domain specific languages, in: 2008 12th International Software Product Line Conference, 2008, pp. 139–148, https://doi.org/10.1109/SPLC.2008.25.

[14] F. Bellifemine, A. Poggi, G. Rimassa, JADE: A FIPA2000 compliant agent development environment, in: Proceedings of the Fifth International Conference on Autonomous Agents, AGENTS '01, ACM, New York, NY, USA, 2001, pp. 216–217, https://doi.org/10.1145/375735.376120.

[15] I. Ayala, M. Amor, L. Fuentes, The sol agent platform: Enabling group communication and interoperability of self-configuring agents in the internet of things, J. Ambient Intell. Smart Environ. 7 (2) (2015) 243–269.

[16] G. Mussbacher, J. Araújo, A. Moreira, D. Amyot, AoURN-based modeling and analysis of software product lines, Softw. Qual. J. 20 (3) (2012) 645–687, http://dx.doi.org/10.1007/s11219-011-9153-8.

[17] S. António, J. Araújo, C. Silva, Adapting the i* framework for software product lines, in: C.A. Heuser, G. Pernul (Eds.), Advances in Conceptual Modeling - Challenging Perspectives, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 286–295.

[18] L. Santos, L. Silva, T. Batista, On the integration of the feature model and PL-AOVGraph, in: Proceedings of the 2011 International Workshop on Early Aspects, EA '11, ACM, New York, NY, USA, 2011, pp. 31–36, http://dx.doi.org/10.1145/1960502.1960509, URL http://doi.acm.org/10.1145/1960502.1960509.

[19] Y. Yu, J.C.S. do Prado Leite, A. Lapouchnian, J. Mylopoulos, Configuring features with stakeholder goals, in: Proceedings of the 2008 ACM Symposium on Applied Computing, SAC '08, ACM, New York, NY, USA, 2008, pp. 645–649, http://dx.doi.org/10.1145/1363686.1363840.

[20] C. Silva, F. Alencar, J. Araújo, A. Moreira, J. Castro, Tailoring an aspectual Goal-Oriented approach to model features ζ, in: Proceedings of the 20th International Conference on Software Engineering & Knowledge Engineering, 2008, pp. 472–477.

[21] ITU-T Z-Series Recommendations, User Requirements Notation (URN) — Language Definition, Tech. rep., Telecommunication Standarization Sector of ITU, 2018.

[22] G. Mussbacher, D. Amyot, Extending the user requirements notation with aspect-oriented concepts, in: R. Reed, A. Bilgic, R. Gotzhein (Eds.), SDL 2009: Design for Motes and Mobiles, Springer, Berlin, Heidelberg, 2009, pp. 115–132.

[23] V.E. Souza, A. Lapouchnian, K. Angelopoulos, J. Mylopoulos, Requirements-driven software evolution, Comput. Sci. 28 (4) (2013) 311–329, http://dx.doi.org/10.1007/s00450-012-0232-2.

[24] L. Montalvillo, O. Díaz, Requirement-driven evolution in software product lines, J. Syst. Softw. 122 (C) (2016) 110–143, http://dx.doi.org/10.1016/j.jss.2016.08.053.

[25] M. Schubanz, A. Pleuss, L. Pradhan, G. Botterweck, A.K. Thurimella, Model-driven planning and monitoring of long-term software product line evolution, in: Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS '13, ACM, New York, NY, USA, 2013, pp. 18:1–18:5, http://dx.doi.org/10.1145/2430502.2430527, URL http://doi.acm.org/10.1145/2430502.2430527.

[26] X. Peng, Y. Yu, W. Zhao, Analyzing evolution of variability in a software product line: From contexts and requirements to features, Inf. Softw. Technol. 53 (7) (2011) 707–721, http://dx.doi.org/10.1016/j.infsof.2011.01.001, URL http://www.sciencedirect.com/science/article/pii/S0950584911000024.

[27] D. Gross, E. Yu, Evolving system architecture to meet changing business goals: An agent and goal-oriented approach, in: Proceedings of the Fifth IEEE International Symposium on Requirements Engineering, RE '01, IEEE Computer Society, Washington, DC, USA, 2001, p. 316, URL http://dl.acm.org/citation.cfm?id=882477.883649.

[28] G. Koliadis, A. Ghose, Relating business process models to goal-oriented requirements models in KAOS, in: A. Hoffmann, B.-h. Kang, D. Richards, S. Tsumoto (Eds.), Advances in Knowledge Acquisition and Management, Springer, Berlin, Heidelberg, 2006, pp. 25–39.

[29] A. Byrski, R. Dreżewski, L. Siwik, M. Kisiel-Dorohinicki, Evolutionary multi-agent systems, Knowl. Eng. Rev. 30 (2015) 171–186, http://dx.doi.org/10.1017/S0269888914000289.

[30] J. Pieter, E.D. de Jong, Evolutionary multi-agent systems, in: International Conference on Parallel Problem Solving from Nature, Springer, 2004, pp. 872–881.

[31] A. Brabazon, M. O'Neill, S. McGarraghy, Introduction to evolutionary computing, in: Natural Computing Algorithms, Springer, Berlin, Heidelberg, 2015, pp. 17–20, http://dx.doi.org/10.1007/978-3-662-43631-8_2, Ch. 1.

[32] H. Joumaa, S. Ploix, S. Abras, G.D. Oliveira, A MAS integrated into home automation system, for the resolution of power management problem in smart homes, Energy Procedia 6 (2011) 786–794.

[33] D.J. Cook, M. Youngblood, S.K. Das, A multi-agent approach to controlling a smart environment, in: Designing Smart Homes, Springer, 2006, pp. 165–182.

[34] Q. Sun, W. Yu, N. Kochurov, Q. Hao, F. Hu, A multi-agent-based intelligent sensor and actuator network design for smart house and home automation, J. Sensor Actuator Netw. 2 (3) (2013) 557–588.

[35] N. Gatti, F. Amigoni, M. Rolando, Multiagent technology solutions for planning in ambient intelligence, in: Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, Vol. 02, WI-IAT '08, IEEE Computer Society, 2008, pp. 286–289, http://dx.doi.org/10.1109/WIIAT.2008.63.

[36] S. Khiat, H. Djamila, A temporal distributed group decision support system based on multi-criteria analysis, Int. J. Interact. Multimedia Artif. Intell. (2019) 1–15, http://dx.doi.org/10.9781/ijimai.2019.03.002, in press.

[37] C. Reinisch, M.J. Kofler, W. Kastner, ThinkHome: A smart home as digital ecosystem, in: 4th IEEE International Conference on Digital Ecosystems and Technologies, IEEE, 2010, pp. 256–261.

[38] P. Egri, Information elicitation for aggregate demand prediction with costly forecasting, Auton. Agents Multi-Agent Syst. 30 (4) (2016) 681–696, http://dx.doi.org/10.1007/s10458-015-9301-9.

[39] T.G. Stavropoulos, G. Koutitas, D. Vrakas, E. Kontopoulos, I. Vlahavas, A smart university platform for building energy monitoring and savings, J. Ambient Intell. Smart Environ. 8 (3) (2016) 301–323.

[40] B. Asare-Bediako, W.L. Kling, P.F. Ribeiro, Multi-agent system architecture for smart home energy management and optimization, in: IEEE PES ISGT Europe 2013, IEEE, 2013, pp. 1–5.

[41] I. Ayala, M. Amor, M. Pinto, L. Fuentes, N. Gámez, iMuseumA: An agent-based context-aware intelligent museum system, Sensors 14 (11) (2014) 21213–21246, http://dx.doi.org/10.3390/s141121213.

[42] C. Wilson, T. Hargreaves, R. Hauxwell-Baldwin, Smart homes and their users: a systematic analysis and key challenges, Pers. Ubiquitous Comput. 19 (2) (2015) 463–476.

[43] N. Gámez, L. Fuentes, Famiware: A family of event-based middleware for ambient intelligence, Pers. Ubiquitous Comput. 15 (4) (2011) 329–339, http://dx.doi.org/10.1007/s00779-010-0354-0.

[44] C. Cetina, P. Giner, J. Fons, V. Pelechano, Autonomic computing through reuse of variability models at runtime: The case of smart homes, Computer 42 (10) (2009) 37–43, http://dx.doi.org/10.1109/MC.2009.309.

[45] H. Khallouki, M. Bahaj, Multimodal generic framework for multimedia documents adaptation, Int. J. Interact. Multimedia Artif. Intell. 5 (4) (2019) 122–127, http://dx.doi.org/10.9781/ijimai.2018.02.009.

[46] I. Ayala, M. Amor, J.M. Horcas, L. Fuentes, Model driven evolution of an agent-based home energy management system, in: New Trends in Intelligent Software Methodologies, Tools and Techniques–Proceedings of the 17th International Conference, SoMeT_18, Granada, Spain, 26–28 September 2018, 2018, pp. 17–30, https://doi.org/10.3233/978-1-61499-900-3-17.

[47] J.M. Horcas, M. Pinto, L. Fuentes, Product line architecture for automatic evolution of multi-tenant applications, in: IEEE 20th EDOC, 2016, pp. 1–10, https://doi.org/10.1109/EDOC.2016.7579384.

[48] S. Hallsteinsen, M. Hinchey, S. Park, K. Schmid, Dynamic software product lines, Computer 41 (4) (2008) 93–95, http://dx.doi.org/10.1109/MC.2008.123.

[49] I. Kumara, J. Han, A. Colman, M. Kapuruge, Runtime evolution of service-based multi-tenant SaaS applications, in: Service-Oriented Computing, Springer, 2013, pp. 192–206.

[50] J.a.B.F. Filho, S. Allier, O. Barais, M. Acher, B. Baudry, Assessing product line derivation operators applied to java source code: An empirical study, in: International Conference on Software Product Line, SPLC, 2015, pp. 36–45, http://dx.doi.org/10.1145/2791060.2791099, URL http://doi.acm.org/10.1145/2791060.2791099.

[51] S. Arora, B. Barak, Computational Complexity: A Modern Approach, Cambridge University Press, 2009.

[52] N. Gamez, L. Fuentes, Architectural evolution of famiWare using cardinality-based feature models, Inf. Softw. Technol. 55 (3) (2013) 563–580, http://dx.doi.org/10.1016/j.infsof.2012.06.012, Special issue on Software reuse and product lines, URL http://www.sciencedirect.com/science/article/pii/S0950584912001152.

[53] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, Experimentation in Software Engineering, Springer, 2012, http://dx.doi.org/10.1007/978-3-642-29044-2, URL https://doi.org/10.1007/978-3-642-29044-2.

[54] M. Mendonca, M. Branco, D. Cowan, S.P.L.O.T.: Software product lines online tools, in: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, OOPSLA '09, ACM, New York, NY, USA, 2009, pp. 761–762, http://dx.doi.org/10.1145/1639950.1640002, URL http://doi.acm.org/10.1145/1639950.1640002.

[55] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, T. Leich, FeatureIDE: An extensible framework for feature-oriented software development, Sci. Comput. Program. 79 (2014) 70–85, http://dx.doi.org/10.1016/j.scico.2012.06.002, Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques, WASDeTT-3 2010, URL http://www.sciencedirect.com/science/article/pii/S0167642312001128.

[56] E. Seidewitz, What models mean, IEEE Softw. 20 (5) (2003) 26–32, http://dx.doi.org/10.1109/MS.2003.1231147.