Reflexive and Evolutional Digital Service Ecosystems with Models at Runtime

Dhaminda B. Abeywickrama
Service and Information Architectures
VTT Technical Research Centre of Finland
Kaitoväylä 1, 90570 Oulu, Finland
dhaminda.abeywickrama@gmail.com

Eila Ovaska
Service and Information Architectures
VTT Technical Research Centre of Finland
Kaitoväylä 1, 90570 Oulu, Finland
eila.ovaska@gmail.com

***Abstract*—Uncertainty in digital service ecosystems (DSEs) can be attributed to several factors like the dynamic nature of the ecosystem and unknown deployment environment, change and evolution of requirements, and co-evolution among ecosystem members. Managing uncertainties in DSEs is challenging, and therefore, novel and solid software architecting methods, techniques and tools are needed. Our research explores the means to handle uncertainties at the software architecture level of DSEs. In this regard, we apply valuable lessons learnt from the models at runtime (M@RT) technique. This paper proposes a novel, dynamic knowledge engineering approach to handle uncertainties in DSEs at runtime using M@RT. This uncertainty handling approach aims to identify and solve two interrelated research problems: *reflexivity* and *evolution* of the ecosystem between the architecture and running system of services. Reflexivity means that the system must have knowledge of its components to make intelligent decisions based on self-awareness. In addition, we provide tool support towards automating reflexivity and evolution. Complex state machines of M@RT that serve as a dynamic knowledgebase are modeled using executable state machines, and generation of software artifacts of the model is performed at execution time. Causal connection is maintained between the runtime models and the running system. We validate and illustrate our approach using a DSE in an ambient-assisted living environment for elderly people.**

Keywords—dynamic knowledge; model-driven development; reflexivity; evolvability; uncertainty; digital ecosystems

I. INTRODUCTION

Today the entire industrial world is transforming from a physical world to a digital one. In this context, a new paradigm has emerged called digital service ecosystems (DSEs), which are open, loosely coupled, domain-clustered, demand-driven, self-organizing agents' environments where each entity is proactive and responsive for its own benefit [1]. In DSEs, the business stakeholders provide the most significant driving factor, which requires digital services (DSs) to handle *uncertainty*. *Uncertainty* in DSEs can be attributed to several factors, such as the dynamic nature of the ecosystem and unknown deployment environment, composition and users; change and evolution of requirements; and co-evolution among ecosystem members. Therefore, managing uncertainties is a challenge, and there is a need for novel and solid software architecting methods, techniques and tools.

In traditional software architecting approaches, uncertainty is addressed by identifying robust solutions at design-time. However, for DSEs which are deployed in highly dynamic environments, identifying solutions at design-time is impractical. It is difficult to anticipate all environmental conditions they will encounter throughout lifetime. Recently, models at runtime (M@RT or runtime models) have been adopted in dynamically adaptive systems (DASs) to cope with uncertainty and resolve them at runtime. A runtime model captures relevant information of the running system for different purposes, which can be part of the system functional features or non-functional features providing quality assurance and analysis [2].

In the past, the concept of uncertainty has been explored extensively by researchers in different scientific disciplines, such as economics [3], physics and psychology. There is extensive literature on model-based system reconfiguration and on self-adaptive systems at the architecture level (e.g., [4], [5]). Also, there are several related works on runtime models to address uncertainties in DASs (e.g., [2], [6], [7], [8]). However, most of these works have been limited to conceptual frameworks or reference models. Also, to the best of our knowledge, none of them provide concrete tool support and case study validation especially in the context of digital service engineering. This provides motivation for this study. The contribution of this paper is twofold. First, this paper proposes a novel, dynamic knowledge engineering approach to handle uncertainties in DSEs at runtime using M@RT. Second, it provides tool support towards automating the engineering approach.

In our approach, M@RT is a *dynamic knowledgebase* [2] that abstracts important information about the system, its operational context, and requirements. The approach aims to identify and solve two interrelated research problems. They are: *reflexivity* and *evolution* of the DSE between architecture and running system of services (e.g., cloud services). Reflexivity means that the system must have knowledge of its components to make intelligent decisions based on self-awareness [9]. To support *reflexivity*, we propose a set of quality-driven, self-adaptive architectural patterns. Complex state machines of M@RT that serve as a dynamic

knowledgebase are modeled using Enterprise Architect's executable state machines [10]. We support *evolution* between architecture and running system of cloud services using generation of software artifacts of the model at execution time. The runtime models and the running system are causally (loosely) connected, so that the both abstractions evolve at the same time. The approach is validated using a DSE, which describes an ambient-assisted living (AAL) environment for elderly people.

The rest of the paper is organized as follows. Section II, provides background information to our work. In Section III, we propose our knowledge engineering approach using M@RT. The case study and the application of the approach are described in Section IV. In Section V, we present key related work, and Section VI concludes this paper.

II. BACKGROUND

A. Definitions

1) Uncertainty:

Uncertainty [11] is a system state of incomplete or inconsistent knowledge so that an adaptive system is unable to know which alternative environmental or system configurations hold at a specific point. It can be caused by missing or ambiguous requirements, false assumptions, unpredictable entities or phenomena in the execution environment, and unresolvable conditions caused by incomplete and inconsistent information. This information can be obtained by potentially imprecise, inaccurate, and unreliable sensors in its monitoring infrastructure [11]. Uncertainty and variability are two separate, yet related notions, and in our research, we aim to deal with uncertainty.

2) Reflexivity and Evolution:

Reflexivity is an important characteristic of a self-managed autonomic system, which means that the system must have knowledge of its components, current status, capabilities, limits, boundaries and interdependencies with other systems and available resources [12]. Also, the system must be aware of its possible configurations and how they affect specific non-functional, quality requirements. In this study, we consider reflexivity as a technique that can be exploited to support evolution of the ecosystem.

By evolution we refer to the ability of the ecosystem to evolve in dynamic situations (e.g., see [13]). An ecosystem is dynamic, evolving all the time as new members, services and value networks emerge [14]. Therefore, to adapt to the needs of the ecosystem, the ecosystem's knowledge management model should evolve too.

B. ADSEng Methodology and Contribution

Previously in [15], we established several characteristics as part of an evaluation framework to compare existing autonomic computing approaches in DSEs. They are: *top-down vs. bottom-up approaches*, *decentralized control*, *self-* properties* and *context-awareness*, *reflexivity*, *quality attributes*, and *validation/case study*. Later, in [9], these characteristics corresponded to several key requirements that are significant in a service engineering method for autonomous DSEs. In this context, we introduced the *ADSEng* methodology [9], which is a novel, systematic service engineering methodology proposed for ecosystem-based engineering of autonomous DSs. This integrated and model-based methodology covers from requirements engineering to architecting, and running systems of DSs. The current paper is based on the *ADSEng* methodology. However, it is a significant step further, presenting a dynamic knowledge engineering approach as well as practical tool support and case study validation.

III. ENGINEERING OF DYNAMIC KNOWLEDGE MODELS

In this section, we describe our dynamic knowledge engineering approach using M@RT to handle uncertainties in DSEs at runtime. First, an overview of the engineering process (see Fig. 1) is provided followed by a description of each step. We use the Sparx Systems' Enterprise Architect (13.0) case tool towards automating the engineering process.

A. Approach Overview

In our approach, a runtime model is a *dynamic knowledgebase* that abstracts important information about the system, its operational context, and requirements. MAPE-K is a reference model introduced by IBM for autonomic control loops [16]. Here, the basic monitor-analyze-plan-execute over a knowledgebase (MAPE-K) architecture [16] of the DSE is extended at the architecture level to use models that evolve, so that the software system is able to better manage uncertainty. This is realized by analyzing new properties about the execution environment and the system itself based on monitoring information collected at runtime. The four key processes in the architecture - *monitor*, *analyze*, *plan* and *execute* – can analyze the system and environmental data to refine and extend the information stored in the runtime models.

Towards supporting *reflexivity*, we propose a set of quality-driven, self-aware and self-adaptive architectural patterns, which are extended MAPE-K loops with quality assurances (see Section III-B). The goal is to use the patterns to create and customize the dynamic knowledge models in different domains. We model the complex state machines of M@RT using Enterprise Architect's executable state machines. *Evolution* between architecture and running system of cloud services is supported using generation of software artifacts of the model at execution time. Causal connection is maintained between runtime models and the running system.

In this manner, the main steps in the dynamic knowledge engineering process are (see steps 1-3 in Fig. 1):

(1) create dynamic knowledge models of requirements and architecture in the example domain (see step 1b in Fig. 1). At the architecture level, this is assisted by a collection of self-aware and self-adaptive architectural patterns (step 1a, Fig. 1);
(2) create an executable state machine artifact to run the knowledge models in architecture and then generate the code and compile it (step 2, Fig. 1);
(3) execute, simulate and validate the dynamic knowledge models (step 3, Fig. 1).

B. Step 1: Create Dynamic Knowledge Models

We use two types of knowledge models at the requirements and architecture levels, respectively. They are: (i) a KAOS-based *goal model* that represents the requirements; and (ii) model-based finite state machines (*FSMs)* at the architecture level. For the goal-based requirements model, a simplified KAOS metamodel is used to represent goals where each goal is associated with a name and a priority. FSMs, which are embodied in the architectural patterns, describe the behavior of the system with respect to the current goals and context. A FSM runtime model can show the current state information of a component during its operation.

1) Architectural Patterns Collection for DSEs:

The main step (see also [9]) for supporting reflexivity is representing the uncertainty factors using *architectural patterns*. The main goal is to use them for creating and customizing the knowledge models in different domains. The patterns have been modeled in Enterprise Architect using *UML templates* (UML activity and state models), which can be instantiated to a particular domain using *UML 2.5 models*. Here the traditional autonomic MAPE-K loops [16] have been extended with quality guarantees to handle uncertainty at the software architecture level.

Both decentralized and centralized feedback loop approaches have been suggested to facilitate autonomic behavior in adaptive systems [17]. We integrate these approaches in the patterns to exploit the benefits of both. With this our approach aims to support both collective adaptation and adaptation by subparts. Although centralized approaches allow global behavior control, they contain a single point of failure and suffer from scalability issues. Conversely, decentralized approaches do not require any a priori knowledge, nor do they contain a single point of failure. In this paper, we describe two main patterns from the collection—the *autonomic DS pattern* and *centralized DS pattern* (see Fig. 2 and Fig. 3). The former applies the decentralized feedback loop approach while the latter applies the centralized approach.

a) Autonomic DS Pattern:

The autonomic DS pattern (see Fig. 2), which is modeled as a UML template, is characterized by the presence of an explicit, external feedback loop (Autonomic Manager / AM) to direct the behavior of the DS, which is the managed element. This pattern exhibits self-* properties, such as self-awareness and self-adaptation. The DS has sensors, effectors and a representation of goals. An AM handles the adaptation of the DS. Several AMs can be associated with the DS, each closing a feedback loop devoted to controlling a specific adaptation aspect of the system, and adding different levels of AMs increases the autonomicity. Here, the novelty is that the traditional autonomic MAPE-K loop is extended with a quality assurance component at the architecture level for DSs. This quality assurance component implements a QoS model that provides QoS guarantees. It corresponds to the dynamic knowledgebase in Fig. 1. The quality assurance component can include three types of runtime models depending on their subject, such as system models (i.e., abstract view of the running system), context models (e.g., environment), and quality requirements of the DS. During the adaptation process,

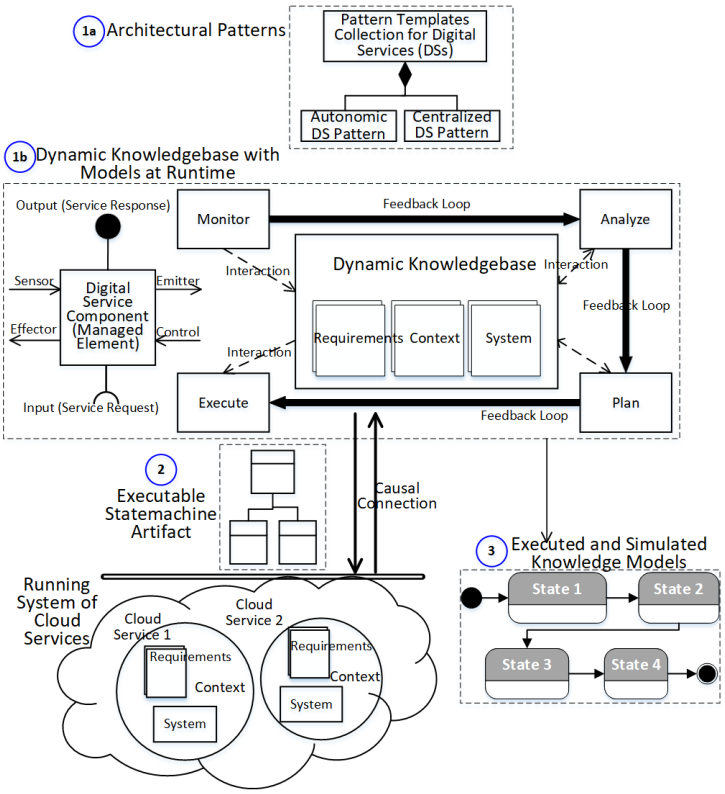

Fig. 1. Dynamic knowledge models-based service engineering.

the MAPE elements of an AM interact with the quality assurance component to obtain or update information about the system states, environment, and quality assurance criteria.

b) Centralized DS Pattern:

This pattern (see Fig. 3) is characterized by a global feedback loop, which manages a higher-level adaptation of behavior of multiple autonomic components (e.g., two DSs in Fig. 3). The adaptation in the centralized DS pattern is handled by a high-level AM called a super AM. Like an AM, a super AM also has a quality-driven, extended MAPE-K adaptation model. This is while the single DSs are able to self-adapt their individual behavior using their own external feedback loops in the AMs (Fig. 3).

C. Step 2: Create Executable State Machine Artifact, Generate and Compile Code

Next, in the knowledge engineering process, we design the *executable state machine artifact*, which is used to generate the code for the model that can be compiled and executed (see step 2 in Fig. 1). This artifact essentially describes the classes and objects involved in the DSs and their AMs in the patterns, their initial properties and relationships. It acts as the binding script that links multiple objects together and determines how these will communicate in a simulation at runtime.

The underlying technique we explore to support *evolution* between the architecture and the running system of cloud services is generation of service software at execution time. In DSEs, which are characterized by a high level of uncertainty, the generation of software needs to happen at runtime as it cannot necessarily be foreseen during design time. In this study

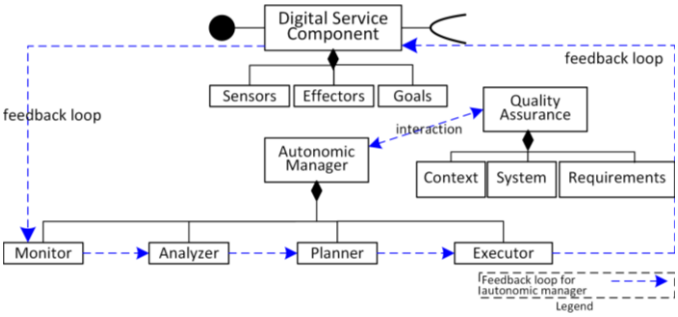

Fig. 2. Autonomic digital service pattern.

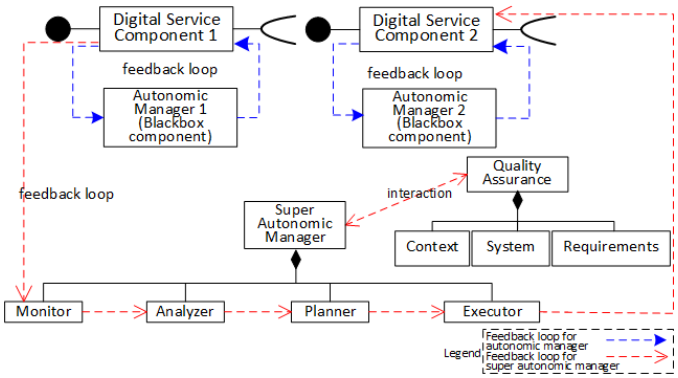

Fig. 3. Centralized digital service pattern.

we use the dynamic *code generation* feature of the executable state machine artifact to support the *evolution* between architecture and running system of services. The automated model transformations in the code generation process support the causal connection between the runtime models and the running system. This means that when the model is modified, the running system is changed correspondingly. Thus, the two abstractions synchronize and evolve at the same time.

D. Step 3: Simulate and Validate Knowledge Models

Finally, we exploit Enterprise Architect's ability to perform simulations with its simulation feature which allows executing the dynamic knowledge models created (step 3, Fig. 1). As the executable state machine executes, the relevant state machine diagrams are displayed where the active state for the instance completing a step is highlighted and the other states remain dimmed. The state machines can be changed at runtime by user interaction. The simulation provides a visual reflection of the real compile code as it is executing. When the execution is completed the generated code can be deployed to the target system. This code can be further modeled by the engineer to derive the *running system of cloud services*.

We use the *debugging* features of the Enterprise Architect environment to validate the complex dynamic knowledge models of state machines. These can be inserting simulation breakpoints, firing waiting triggers, tools to pause and run a simulation, and tools to examine local variables and the call stack. Thus, it allows us to verify the correct behavior of the generated code.

IV. Applying the Approach

After presenting the proposed approach, now we describe how we have applied it to the AAL case study. The scope of the knowledge engineering process is large. Therefore, as applied next using the case study, this paper focuses on step 1 and step 2 to support reflexivity and evolution of DSEs between architecture and running system of cloud services. Simulating and validating of knowledge models (step 3) in the case study will be addressed in a future paper.

A. Problem Domain and Case Study

The case study describes a DSE in digital health revolution, which provides an AAL environment for elderly people [9] (see Fig. 4). Advanced smart homes depend on adaptivity to function properly [9]. This can be associated with several uncertainty factors; e.g., sensors or devices can fail, and the behavior of the elderly person him/herself can be highly uncertain. Therefore, in such uncertainty situations, the system needs to satisfy the requirement in some other way. In AAL, the DSE includes two main cloud-based services: (i) a monitoring and security service and (ii) a diagnosis service. The monitoring and security service utilizes different monitoring devices to analyze the elderly person's activities and provides security services. This service can include several *supporting cloud services*; for example, several sensor-based monitoring services and an elderly care security service. The service providers can be different businesses that provide sensor services, and an elderly care security service. Meanwhile, the diagnosis cloud service is used by a doctor or another person (e.g., a nurse) to make a diagnosis based on monitored data. This service can include several supporting cloud services provided by several service providers; for example, electric health records, health insurance and clinical information systems.

In this context, we describe a scenario of an elderly person called *Peter*: *Peter is 74 years old, and suffers from nocturnal epileptic seizures; and he is also an alcoholic. The main goal for Peter is to maintain health (i.e., reduce the triggering of seizures), which can be dependent on two factors: high alcohol level and lack of sleep due to disturbances to sleep pattern. Several devices can be associated with several context dimensions, such as: a waist-worn fall detector; an intelligent bed with epileptic sensor to monitor seizures; activity monitors to monitor sleep level; an intelligent cellar with RFIDs to monitor the locations of alcohol containers; intelligent mug with sensors to monitor the alcohol level consumed.*

The devices—the intelligent cellar and intelligent mug—are supporting cloud services. They are connected to the ambient-assisted home hub unit for a more high-level handling of sensor information and adaptation of the DSs. Thus, there are two supporting cloud services: `iCellar` (i.e., RDF tracking sensor cloud service in Fig. 4) and `iMug`, and the `ambient-assisted home hub unit` can be considered as a high-level service. Here, a composite DS can be the orchestration of the `iCellar` and `iMug` DSs. The goal of the `ambient-assisted home hub unit` service is to monitor the alcohol volume of the containers in the cellar, and

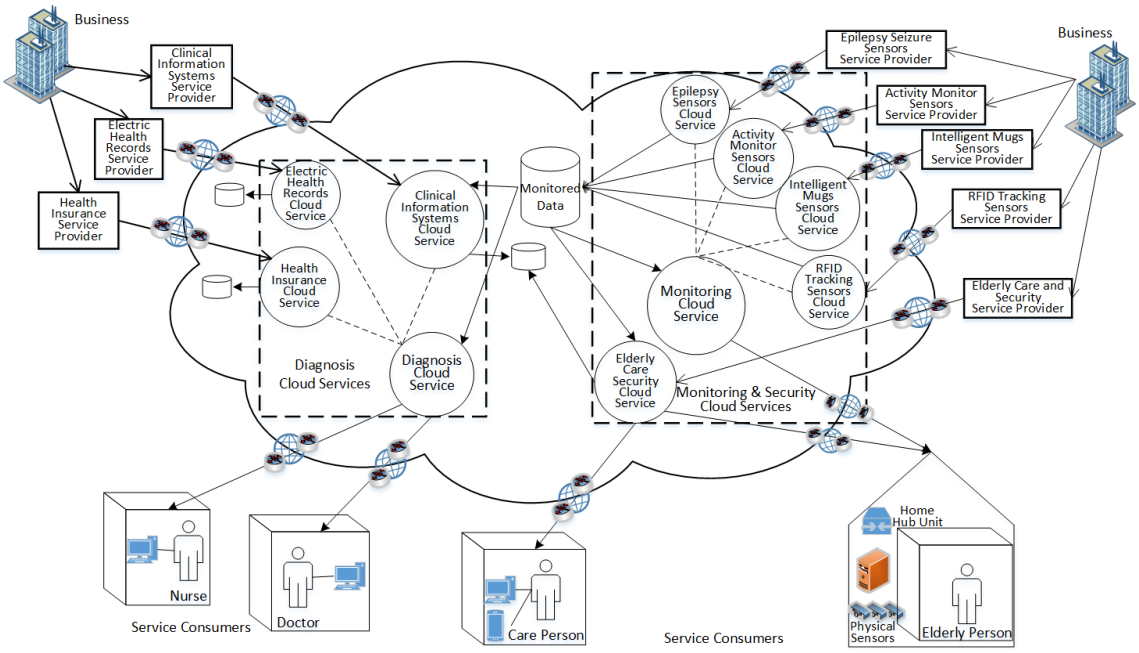

Fig. 4. Case study: ambient-assisted living (AAL) digital service ecosystem.

alcohol volume consumed from the mug, so the maximum alcohol intake level is not exceeded. In this example, we assume that there is a single container in the `iCellar`. Table 1 summarizes the main goals of these cloud services and their associated uncertainty factors.

B. *Applying the Approach to Case Study*

1) *Create Dynamic Knowledge Models:*

a) Requirements Model:

In the AAL case study, we use KAOS-based *goal models* to represent the requirements, i.e., goals (functional) and softgoals (non-functional) of the scenario. Fig. 5 provides an excerpt of a goal graph created to mitigate uncertainties. There the top-level goal is refined as a goal lattice in which branches and goals are refined into expectations and requirements. The main goal for Peter is to maintain his health by avoiding triggering of seizures. In the goal model, KAOS obstacles are used to represent the uncertainty factors, e.g., Peter's behavior. More specifically, it can happen that Peter forgets to control alcohol intake. That is, it is uncertain whether Peter will avoid high alcohol intake; he could forget to control drinking and the effect could mean he exceeds alcohol intake, becomes intoxicated and eventually unhealthy.

As stated in Section III-B, a simplified KAOS metamodel is used to represent goals where each goal is associated with a name and a priority. The goal priorities can be affected by context changes, thereby affecting tasks execution by the user (e.g., Peter) or entity (e.g., `iCellar`, `iMug` or `ambient-assisted home hub unit`) at runtime. The system depending on the goal priority and the relation between the goals weighs the requirements during task execution and selects a set of tasks to be performed. This goal configuration graph for the `ambient-assisted home hub unit` service is shown in Fig. 6. The difference between the left and right configurations is in the priority level of the `low alcohol consumption` softgoal where the right configuration has a higher priority.

TABLE I. GOALS AND ASSOCIATED UNCERTAINTY FACTORS OF SUPPORTING CLOUD SERVICES

Supporting Cloud Service	Goals	Uncertainties
Intelligent cellar (`iCellar`)	Maintain alcohol volume consumed from container	Alcohol containers can get misplaced or lost
Intelligent mug (`iMug`)	Maintain alcohol volume consumed from mug	Alcohol content can be spilled; other drinking objects can be present
`Ambient-assisted home hub unit`	Ensure that maximum alcohol intake level is not exceeded; Raise an alarm if it exceeds a certain threshold level	Alcohol intake level is exceeded which can cause Peter to get intoxicated and become unhealthy

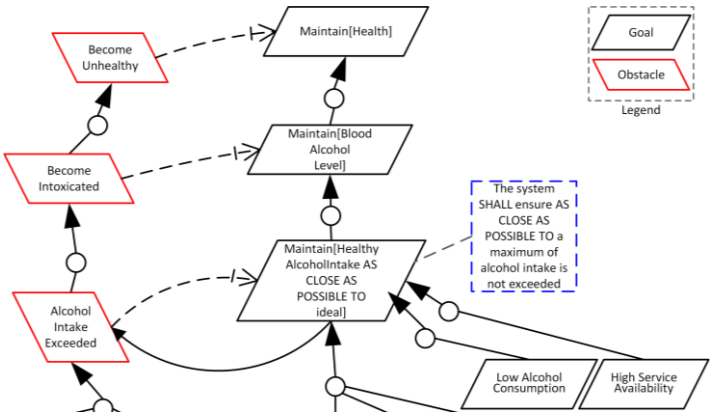

Fig. 5. Requirements: goal model to mitigate uncertainties in AAL.

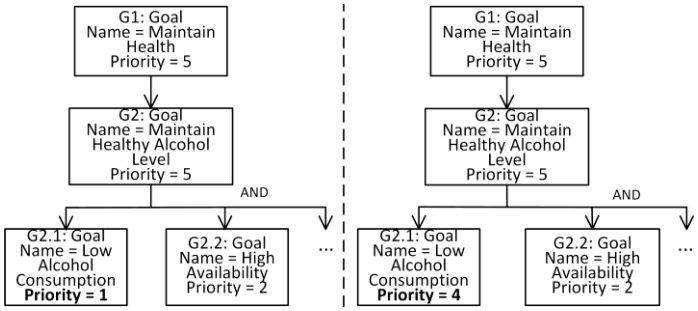

Fig. 6. Two goal configurations for the requirements of the ambient-assisted home hub unit service.

b) Architecture Model:

We now describe the architectural models realized applying our approach to simulate several feedback loop structures in the `iCellar-iMug` DSs composition of the AAL case study (see Fig. 7). These models are created using the patterns presented in Section III-B. In this study, UML 2.5 activity models and state machines have been used as the primary notation to model the behavior of the loops. A feedback loop provides the interplay between flow (control or data) and actions on the flows.

One of our main motivations of the present research is to address uncertainty. To this end, the approach and the simulation scenario add unexpected context information to model uncertainty using state machines. To address uncertainty, the monitoring needs to be context-aware. The analysis phase then reasons using new (unknown) context information and interprets how the goals are affected, thus planning for the system to cope with such changes.

Managed Elements and Autonomic Managers: There are two managed elements (DSs) in the AAL case study example: `DigitalService_ICellar` and `DigitalService_IMug` (Fig. 7). The two AMs–`AutonomicManager_ICellar` and `AutonomicManager_IMug`—close separate, decentralized feedback loops to handle the adaptation of alcohol volume in container and alcohol volume in mug (and spilled volume), respectively. Also, there is a high-level AM called a super AM (`SuperAutonomicManager_AmbientAssistedHome HubUnit`) that closes a separate, centralized feedback loop. Here, this super AM handles the adaptation of alcohol volume in both the `iCellar` and the `iMug`. As stated in Section III-B, a super AM can manage the adaptation of multiple SCs. This example implements and integrates both decentralized and centralized feedback control loop techniques using the two patterns—`autonomic DS pattern` and `centralized DS pattern`. In the example, the runtime model is now a goal model with constraints, an environment model, and an initial behavior model in the form of FSMs.

The resolving of uncertainties at runtime by the different phases (i.e., monitoring, planning, analyzing and executing) of the extended MAPE-K loop architecture is explained next.

Monitoring: The monitoring process has two main tasks: monitoring and updating of runtime models. It measures raw

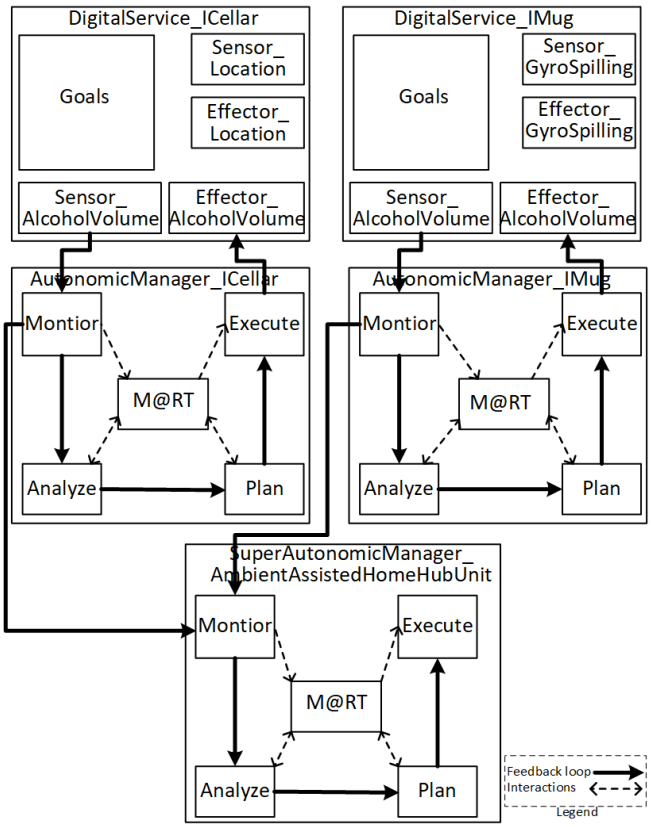

Fig. 7. AAL Case study: digital services and autonomic managers.

data through sensors (e.g., collect alcohol volume in `Monitor` of `AutonomicManager_ICellar`, Fig. 8) about the current state and/or occurring events of the system, the context, and the requirements (goals). It checks for changes of goals, which can be changed by the user or the system itself. The monitoring process updates the runtime model that represents knowledge about the state, context, and requirements (change of goals).

Analyzing: There are two main tasks in this process: (i) it gathers runtime models and interprets data collected by the monitoring process against goals and constraints; and (ii) detects system and environmental changes that may need adaptation. The analyze process decides that a behavior adaptation is required to satisfy the detected goal changes. In the example, this can be gathering and interpreting alcohol volume data from the runtime model and checking against the goals to detect changes that require adaptation (see `Analyze` of `AutonomicManager_ICellar`, Fig. 8).

Planning: The planning process reads the runtime model enhanced by the analysis process. Then some reasoning is performed to identify how the running system should be best adapted to changes of the system, context, and requirements. The planned changes can be in the form of a runtime model. In the example, the `high availability` softgoal of the `ambient-assisted home hub unit` service (`SuperAutonomicManager_AmbientAssistedHome HubUnit`) has a much high priority than the softgoal `low alcohol consumption` (see Fig. 6, left). Therefore, the planning step for that component can generate a new

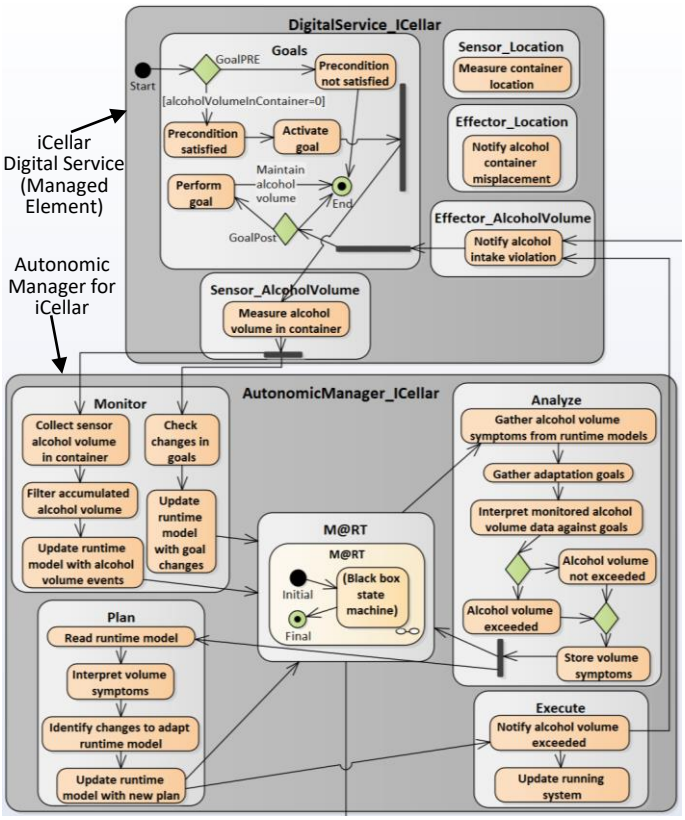

Fig. 8. Modeling iCellar service and autonomic manager in AAL.

behavioral model or adapt the existing one (see M@RT state machines in Fig. 9, top). Here, there is a reduced alcohol consumed level safety margin where the critical behavior is entered only if the alcohol level is greater than 80. The `ambient-assisted home hub unit` continues to perform its tasks according to the specialized state machine until goals changed by the user or system itself. Let us assume that the goals change to the situation as depicted in the second configuration (see Fig. 6, right). Now the `low alcohol consumption` goal has a higher priority. The monitoring step of the loop senses this change of goals, and updates it in the runtime model. The analyze activity decides that a behavior adaptation is necessary and the planning step tries to fulfill the new constraints. In this case, the MAPE loop will generate or adapt the existing state machine (see M@RT' state machines in Fig. 9, bottom). The new behavioral model uses a higher safety margin for alcohol consumed level (critical stage is entered when it is greater than 90). In addition, it introduces a new state called `Error` to handle the critical behavior of the state machine by adding additional operations (functions) through error handling extensions in the `ambient-assisted home hub unit` service.

Executing: The Execution process directly applies a set of changes for the running system stored in some runtime models by the planner. In this example, it can be notifications sent on over alcohol consumption and to control drinking.

2) Create Executable State Machine Artifact, Generate and Compile Code:

As mentioned in Section III-C, after creating the classes

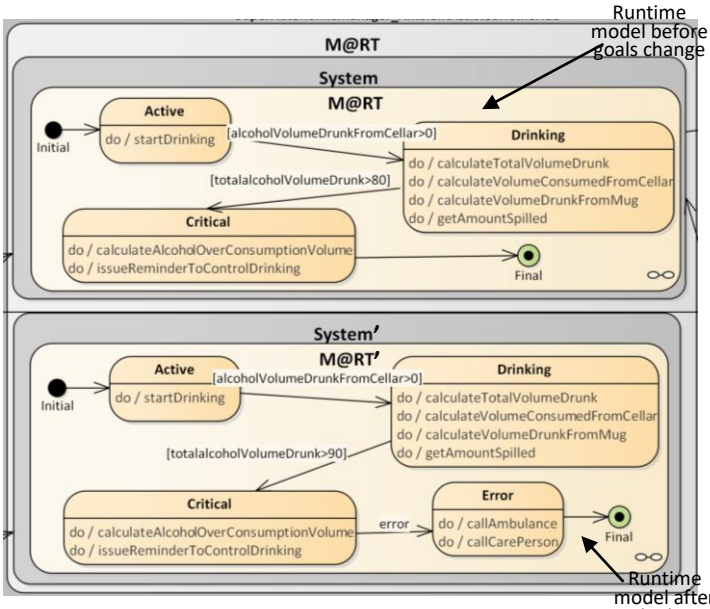

Fig. 9. M@RT of the ambient-assisted home hub unit service in AAL.

and state machines on knowledge models then we design the executable state machine artifact (`AALCaseStudy_ExecutableArtifact`, Fig. 10). This artifact describes the classes and objects involved in the example (i.e., the DSs and their AMs in the patterns). They can also describe their initial properties and relationships. We define initial state of instance by assigning property values to the class attributes. For example, see property values assigned to `DS_ICellar`, `DS_IMug` and `SAM_AmbientAssistedHomeHub`. Also, we define how each property can reference other properties by defining relationships based on the class model that they are instances of.

Afterwards, using the executable state machine artifact created in the example, we generate the code in Java and compile it. See Fig. 10 for system output with code successfully generated for all classes. The generation of code from the executable state machine artifact supports causal connection between the runtime models and the running system, thus both evolving at the same time. Next, the compiled code can be executed, simulated and validated using the simulation and debugging features of Enterprise Architect.

V. Related Work and Discussion

A key research area related to our study is *model-based system reconfiguration*. *Rainbow framework* [4] is a seminal work on architecture-based self-adaptation. It extends architecture styles to provide reusable infrastructure with mechanisms to specialize the infrastructure to the needs of specific systems. Braberman et al. [5] proposed a three-layered reference architecture called *MORPH* for architecture-based adaptation that involves runtime change of system configuration and behavior update. Meanwhile, Morin et al. [18] proposed an approach that combines aspect-oriented and model-driven techniques to limit the number of artifacts needed to realize dynamic variability. However, these works

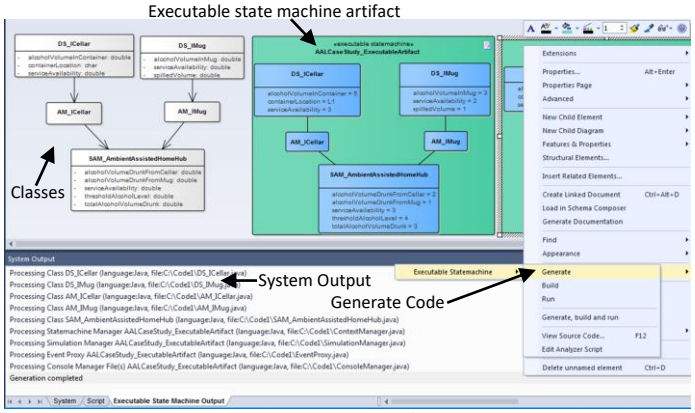

Fig. 10. Executable state machine artifact and code generation.

are neither based on M@RT explicitly nor target digital service ecosystems.

Maes [19] described one of the pioneering works on *reflection* with respect to object-oriented programming languages. The author defines a computational system causally connected to its domain and a change in its domain is reflected on it and vice versa. The *Twin Peaks* model [20] deals with a similar notion to reflexivity proposed here. It is an iterative process that develops progressively more detailed software requirements and architectural specifications concurrently. *Requirement reflection* [21] supports runtime representation of requirements by making requirements available as runtime objects for DASs. Bencomo [21] classifies uncertainty and adaptations that a self-adaptive system need to face. There to deal with uncertainty, goal-oriented requirements modeling has been extended with the RELAX language. The goal of their work is to manage uncertainty primarily at the requirements level.

In [11], the authors have proposed a taxonomy of potential sources of uncertainty and techniques for mitigating them in the requirements, design, and execution phases of DASs. In [22], the authors present a *goal-based modeling* approach to develop requirements of an adaptive system with environmental uncertainty. While our technique of handling uncertainties using a KAOS-based goal-model follows [22], our work differs from [22] in that we handle uncertainty using M@RT at the architecture level as well.

Adaptation patterns have been explored as a technique to provide self-adaptation in several works (e.g., [23]). However, the novelty is the patterns defined here for DSEs contain a key quality assurance component for providing quality guarantees in addition to the adaptation handling MAPE elements.

There are several works (e.g., [2], [6], [7], [8]) that use *runtime models* as a technique to manage uncertainties and provide assurance in DASs. The authors in [2], [6] propose extensions to the MAPE-K loop architecture with runtime models to cope with uncertainty at runtime. In [6], the authors present a conceptual reference model called *MAPE-MART*, which extends the traditional MAPE-K model with quality assurance mechanisms for self-adaptation. The notion of dynamic knowledgebase was originally motivated by the work in [2] where the authors identify a runtime model as a dynamic knowledgebase. However, the MAPE-K extensions proposed in their approach are only at the conceptual level.

DYNAMICO (Dynamic Adaptive, Monitoring and Control Objectives model) [7] is a reference model for engineering context-based self-adaptive software composed of feedback loops. It aims to guarantee the coherence of adaptation mechanisms with respect to changes in adaptation goals and monitoring mechanisms. Tamura et al. [8] present a proposal for including software validation and verification (V&V) operations explicitly in MAPE-K loops for achieving software self-adaptation goals. They discuss runtime V&V enablers, i.e., requirements at runtime, models at runtime, and dynamic context monitoring, for providing effective support to materialize V&V assurances for self-adaptation.

To summarize, extensive literature exist on model-based system reconfiguration and on self-adaptive systems in the architecture level (e.g., [4], [5]). Also, there are several approaches on runtime models to address uncertainties in DASs (e.g., [2], [6], [7], [8]). However, most of these have been limited to conceptual frameworks. Also, to the best of our knowledge, none of them provide concrete tool support and case study validation especially in the context of DSEs. We have presented an approach for supporting M@RT in DSEs that provide reflexivity and evolution.

VI. Conclusions and Future Work

In this paper, we presented a dynamic knowledge engineering approach to handle uncertainties in DSEs at runtime using the M@RT technique. The approach aims to solve two interrelated research problems, i.e., reflexivity and evolvability of the ecosystem between the architecture and the running system of services. Complex state machines of M@RT that serve as a dynamic knowledgebase have been modeled using executable state machines, and the generation of software artifacts has been performed at execution time. Causal connection has been maintained between the runtime models and the running system to support evolution. We presented an example involving a DSE in an AAL environment for elderly people for validating the approach.

The aim of our work is not to enhance model-driven architecture but to create an approach that can be applied to digital service engineering and is based on service architecture. The approach needs to be easy to understand as the target audience is software engineers and not specialists in autonomic computing. Currently, by using the capability of Enterprise Architect, our approach can automate the forward synchronization of the causal connection between the architecture and running system. At present, we are exploring techniques for supporting the backward synchronization from running system of cloud services to the architectural models. Our future work will include completing the building of the collection of quality-driven adaptation patterns. Also, so far, our modeling has considered only a single user in the ecosystem (i.e., elderly person). This needs to be applied to multiple users of the ecosystem to further validate our approach. Finally, in order to provide a true assessment of our knowledge engineering approach, developing industrial case studies with empirical cases is significant.

ACKNOWLEDGMENT

This work was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme. This research has also been supported by a grant from Tekes—the Finnish funding agency for technology and innovation, and VTT as part of the Digital Health Revolution Programme.

REFERENCES

[1] H. Boley and E. Chang, "Digital ecosystems: principles and semantics," in *Proceedings of the International Conference on Digital EcoSystems and Technologies (DEST '07)*, Cairns, Australia, 2007, pp. 398-403.

[2] H. Giese, N. Bencomo, L. Pasquale, A. J. Ramirez, P. Inverardi, S. Wätzoldt and S. Clarke, "Living with uncertainty in the age of runtime models," in *Models@run.time: Foundations, Applications, and Roadmaps*, N. Bencomo *et al*, Eds. Cham: Springer, 2014, pp. 47-100.

[3] J. Laffont, *The Economics of Uncertainty and Information*, The MIT Press, 1989.

[4] D. Garlan, S. Cheng, A. Huang, B. Schmerl and P. Steenkiste, "Rainbow: architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, pp. 46-54, 2004.

[5] V. Braberman, N. D'Ippolito, J. Kramer, D. Sykes and S. Uchitel, "MORPH: A reference architecture for configuration and behaviour self-adaptation," in *Proceedings of the 1st International Workshop on Control Theory for Software Engineering*, Bergamo, Italy, 2015, pp. 9-16.

[6] B. C. Cheng, K. Eder, M. Gogolla, L. Grunske, M. Litoiu, H. Müller, P. Pelliccione, A. Perini, N. Qureshi, B. Rumpe, D. Schneider, F. Trollmann and N. Villegas, "Using models at runtime to address assurance for self-adaptive systems," in *Models@run.Time*, N. Bencomo *et al*, Eds. Springer International Publishing, 2014, pp. 101-136.

[7] N. M. Villegas, G. Tamura, H. A. Müller, L. Duchien and R. Casallas, "DYNAMICO: A reference model for governing control objectives and context relevance in self-adaptive software systems," in *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, R. de Lemos *et al*, Eds. Springer, 2013, pp. 265-293.

[8] G. Tamura, N. M. Villegas, H. A. Müller, J. P. Sousa, B. Becker, G. Karsai, S. Mankovskii, M. PezzÃ¨, W. Schäfer, L. Tahvildari and K. Wong, "Towards practical runtime verification and validation of self-adaptive software systems," in *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, R. de Lemos *et al*, Eds. Springer, 2013, pp. 108-132.

[9] D. B. Abeywickrama and E. Ovaska, "ADSEng: a model-based methodology for autonomous digital service engineering," in *Proceedings of the 8th International ACM Conference on Management of Digital EcoSystems (MEDES'16)*, Hendaye, France, 2016, pp. 34-42.

[10] Sparx Systems Enterprise Architect. *Executable State Machines* [Online]. Available: http://www.sparxsystems.com/resources/user-guides/simulation/executable-state-machines.pdf. Last Accessed: 6/7/2017.

[11] A. J. Ramirez, A. C. Jensen and B. H. C. Cheng, "A taxonomy of uncertainty for dynamically adaptive systems," in *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'12)*, Zurich, Switzerland, 2012, pp. 99-108.

[12] H. A. Müller, L. O'Brien, M. Klein and B. Wood, "Autonomic computing,", Carnegie Mellon University, USA, Rep. Report No.: CMU/SEI-2006-TN-006 2006.

[13] G. Briscoe and P. De Wilde, "Digital ecosystems: evolving service-orientated architectures," in *Proceedings of the 1st Bio-Inspired Models of Network, Information and Computing Systems Conference (BIONETICS 2006)*, Madonna di Campiglio, 2006, pp. 1-6.

[14] A. Immonen, E. Ovaska, J. Kalaoja and D. Pakkala, "A service requirements engineering method for a digital service ecosystem," *Service Oriented Computing and Applications*, vol. 10, no. 2, pp. 151-172 2016.

[15] D. B. Abeywickrama and E. Ovaska, "A survey of autonomic computing methods in digital service ecosystems," *Service Oriented Computing and Applications*, vol. 11, no. 1, pp. 1-31 2017.

[16] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41-50 2003.

[17] T. Haupt, "Towards mediation-based self-healing of data-driven business processes," in *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, Zurich, Switzerland, 2012, pp. 139-144.

[18] B. Morin, O. Barais, G. Nain and J. M. Jezequel, "Taming dynamically adaptive systems using models and aspects," in *Proceedings of the IEEE 31st International Conference on Software Engineering*, 2009, pp. 122-132.

[19] P. Maes, "Concepts and experiments in computational reflection," in *Proceedings of the Object-oriented Programming Systems, Languages and Applications Conference (OOPSLA'87)*, Orlando, Florida, USA, 1987, pp. 147-155.

[20] B. Nuseibeh, "Weaving together requirements and architectures," *Computer*, vol. 34, no. 3, pp. 115-117, mar 2001.

[21] N. Bencomo, "Requirements for self-adaptation," in *Generative and Transformational Techniques in Software Engineering IV*, R. Lämmel, J. Saraiva and J. Visser, Eds. Berlin Heidelberg: Springer, 2013, pp. 271-296.

[22] B. C. Cheng, P. Sawyer, N. Bencomo and J. Whittle, "A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty," in *Model Driven Engineering Languages and Systems*, A. Schürr and B. Selic, Eds. Berlin Heidelberg: Springer, 2009, pp. 468-483.

[23] D. B. Abeywickrama, N. Hoch and F. Zambonelli, "Engineering and implementing software architectural patterns based on feedback loops," *Scalable Computing: Practice and Experience*, vol. 15, no. 4, pp. 291 2014.