DOI 10.12694/scpe.v15i4.1052
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 15, Number 4, pp. 291–307. http://www.scpe.org 

# ENGINEERING AND IMPLEMENTING SOFTWARE ARCHITECTURAL PATTERNS BASED ON FEEDBACK LOOPS

DHAMINDA B. ABEYWICKRAMA*, NICKLAS HOCH †, AND FRANCO ZAMBONELLI‡

**Abstract.** A highly decentralized system of autonomous service components consists of multiple and interacting feedback loops which can be organized into a variety of architectural patterns. The highly complex nature of these loops make engineering and implementation of these patterns a very challenging task. In this paper, we present SimSOTA—an integrated Eclipse plug-in to architect, engineer and implement self-adaptive systems based on our feedback loop-based approach. SimSOTA adopts model-driven development to model and simulate complex self-adaptive architectural patterns, and to automate the generation of Java-based implementation code. The approach is validated using a case study in cooperative electric vehicles.

**Key words:** architectural patterns, autonomic systems, software engineering, self-adaptive systems, simulation, model-driven development, Eclipse plug-ins

**AMS subject classifications.** 68N99

## 1. Introduction.

Software systems are becoming increasingly complex and decentralized, and called to function in highly dynamic, open-ended environments. Thus, developing, deploying and managing these systems with the required level of reliability and availability have been very challenging. As a consequence, new software engineering methods and tools are required to make these systems *autonomic* [10], i.e. capable of automatically tuning their behaviour and structure in response to the dynamics of the operational environment in a *self-aware* and *self-adaptive* way.

In the context of the ASCENS project (Autonomic Service Component Ensemble, *www.ascens-ist.eu*), various models and mechanisms have been studied to integrate autonomic features in large-scale service systems, both at the level of service components and service ensembles. In particular, the SOTA ("State Of The Affairs") model has been defined [2] as a general goal-oriented framework for analysing the self-awareness and self-adaptation requirements of adaptive systems and for supporting the design of autonomic service ensembles.

To achieve such autonomic capability, *feedback loops* are required inside the system. The SOTA framework accordingly defines a catalogue of *self-adaptive patterns*, defining the many possible ways according to which feedback loops can be organized [6, 13]. However, a highly decentralized autonomic system may consist of a large number of service components, and multiple autonomic managers that close multiple and interacting feedback loops. Therefore, in addition to the catalogue, a framework such as SOTA should also provide solid engineering tools to actually support the process of designing and implementing autonomic systems that integrate such patterns. Several works like [8, 11, 12, 14, 15, 16] have addressed the need to make feedback loops explicit or first-class entities. However, little attention has been given to providing solid tool support for their engineering and implementation.

In this paper, we present SimSOTA—an integrated Eclipse plug-in we have developed to architect, engineer and implement self-adaptive systems based on our feedback loop-based approach. The SimSOTA integrated plug-in contains several plug-ins grouped together (i.e. a simulation plug-in and plug-ins for transformation). Our model-driven engineering approach integrates both decentralized and centralized feedback loop techniques in order to exploit their benefits. The SimSOTA plug-in facilitates (1) the modelling, simulating and validating of self-adaptive systems based on the SOTA feedback loop-based approach; and (2) the automatic generation of pattern implementation code in Java using transformations. We validate and assess our approach and plug-in using a case study in cooperative electric vehicles (e-mobility) [9].

In our previous work [4, 5], we presented early results of SimSOTA as a simulation plug-in for modelling and simulating patterns. However, the initial plug-in did not facilitate any application-independent instantiation of models or implementation of complex feedback loops, as supported by the current integrated Eclipse plug-in

*Fraunhofer FOKUS, Berlin, Germany (dhaminda.abeywickrama@gmail.com).
†Corporate Research Group, Volkswagen, Wolfsburg, Germany (nicklas.hoch@volkswagen.de).
‡Dipartimento di Scienze e Metodi dell'Ingegneria, Universit degli studi di Modena e Reggio Emilia, Italy (franco.zambonelli@unimore.it).

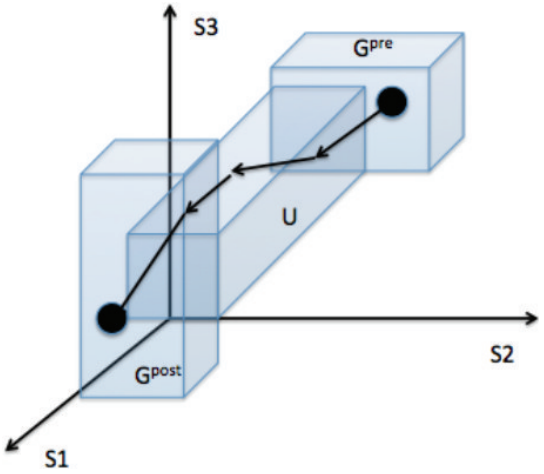


FIG. 2.1. *The trajectory of an entity in the SOTA space [2].*

of SimSOTA. The current paper extends [1] with more results of patterns engineered and implemented. This paper is extended with the help of a detailed case study in e-mobility showing the feasibility of the approach in complex scenarios. An additional pattern called the `Parallel AMs SC` pattern is introduced. The SOTA patterns profile created to represent our feedback loop-based approach is also introduced here. Another key addition is a detailed discussion of our approach against other key works.

The rest of the paper is organized as follows. In Section 2, we present the SOTA conceptual model and patterns catalogue, and the motivation to provide tool support. Section 3 describes the domain independent models created to facilitate the engineering process of the patterns. The case study used to derive platform-specific models is explained in Section 4. In Section 5, the domain-specific models created for the case study at both platform-independent and platform-specific levels are discussed. Section 6 provides a discussion of our approach and related work. Section 7 concludes this paper.

**2. SOTA Patterns Catalogue and Tool Support.** The SOTA model defined in the context of the ASCENS project considers that the actual execution of service components (SCs) and ensembles (SCEs) can be assimilated to that of a dynamical system: during its execution, it traces a trajectory in a virtual $n$-dimensional state space whose dimensions represent the relevant parameters of the execution, and where the goal $G_{post}$ of system execution is reaching a given "state of the affair", i.e. a specific area in the state space, starting from a given initial area $G_{pre}$ (the precondition for activating the goal), and possibly following a bounded trajectory $U$. Figure 2.1 represents these concepts graphically. Accordingly to such perspective, a system is considered self-aware if it can autonomously recognize its current position and direction of movement in the state space, and self-adaptation means that the system is able to dynamically direct its trajectory within $U$ and towards $G_{post}$ despite the fact that the dynamics of the operational environment tend to move it away from it. For a more formal description of these notions of SOTA refer to [2].

The SOTA modelling approach is key to understanding and modelling self-awareness and adaptation, and for checking the correctness of the specification [2, 3]. However, when a designer considers the actual architectural design of the system, it is important to identify which architectural schemes need to be chosen for the individual SCs and SCEs. To this end, To achieve such autonomic capability, *feedback loops* are required inside the system. That is, components and ensemble should somehow integrate control systems (in the form of so called "autonomic managers") to detect the current trajectory of the executing system, and when needed correct it so that specific regions of the space can be reached which correspond to specific application goals. The SOTA framework accordingly defines a catalogue of *self-adaptive patterns*, defining the many possible ways according to which feedback loops can be organized [6, 13], and supporting designers in their choices.

The SimSOTA tool completes the SOTA framework by supporting the actual process of designing and implementing such patterns, i.e. properly structured systems of SCs, SCEs, and their associated autonomic

managers. More in particular, SimSOTA (developed using IBM Rational Software Architect Simulation Toolkit 8.0) is an Eclipse plug-in aimed at providing an integrated environment for the architect to engineer and implement SOTA patterns in their respective domains. It contains several plug-ins grouped together, i.e. a simulation plug-in and plug-ins for transformation.

A key goal of SimSOTA simulation plug-in is to facilitate the engineering (modelling, simulating and validating) of complex self-adaptive systems based on feedback loops. It aims to provide a set of pattern templates (custom profile applied) for all the key SOTA patterns. This is to facilitate general-purpose and application-independent instantiation of models for complex systems based on feedback loops. The SimSOTA plug-in can also be used to simulate and animate the patterns to better understand their complex and dynamic model behaviour. Also, the feedback loop models and their interplay can be validated to detect errors early and to check whether the specified behaviour works as intended.

On the other hand, the transformation plug-ins of SimSOTA are aimed at facilitating the automatic generation of implementation code in Java for the patterns. SimSOTA applies model transformations to automate the application of UML-based architectural design patterns and generate infrastructure code for the patterns using Java semantics. Here, model transformation techniques are applied as a bridge to enforce correct separation of concerns between two design abstractions, i.e. UML-based patterns and their Java implementations. Two main benefits of applying transformations here are improving the quality and productivity of patterns development, and easing system maintenance and evolution for the patterns engineer.

**3. Domain-Independent Pattern Models.** The models and code developed using the SimSOTA tool can be at the domain-independent or domain-specific levels. This section discusses the domain-independent pattern template models created to facilitate the model-driven engineering process of the patterns at the platform-independent UML and platform-specific Java levels. This is discussed using three key SOTA patterns of the catalogue, i.e. `Autonomic SC pattern`, `Parallel AMs SC pattern` and `Centralized SCE pattern`.

**3.1. Notion of Feedback Loop used in SOTA.** The notion of feedback loops explored in our patterns extends the well-established IBM's MAPE-K adaptation model [10] with multiple, interacting loops (see [4] for more details). MAPE-K (i.e. monitor, analyse, plan, execute over a knowledge base) is a reference model introduced by IBM for autonomic control loops [10]. Advanced software systems that are often highly decentralized require the modelling of multiple interacting control loops. Multiple feedback loops can coordinate and support adaptation using two basic mechanisms: *inter-loop* and *intra-loop* coordination [15]. Intra-loop coordination is provided by multiple sub-loops within a single feedback loop, which allows the various phases of MAPE-K in a loop to coordinate with one another. In contrast, inter-loop coordination supports the coordination of adaptation across multiple control loops. These inter-loops can interact using three basic mechanisms: *hierarchy*, *stigmergy* and *direct interaction* [4, 14].

Both decentralized and centralized feedback loop approaches have been suggested to facilitate autonomic behaviour in adaptive systems [4, 19]. We integrate these approaches in order to exploit the benefits of both. Although centralized approaches allow global behaviour 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.

We have developed a UML activity-based custom profile (`SOTA patterns profile`) to model the different elements of the SOTA feedback loop notion applied in the patterns. A custom UML profile introduces a set of stereotypes that extends the existing meta-model of UML to a specific area of application. The `SOTA patterns profile` extends and customizes the UML meta-model for activity diagrams described in the UML 2.4.1 infrastructure and superstructure specifications. It contains several stereotypes which are used by the transformation plug-ins when generating implementation code for the patterns.

SOTA identifies and classifies patterns based on the above described dimensions, and defines a taxonomy that is helpful for the engineering of multiple and possibly interacting feedback loops, in particular for choosing among a variety of feedback loop compositions.

**3.2. UML Template Models for the Patterns.** We provide a set of UML pattern templates for all the key SOTA patterns (e.g. `Primitive SC`, `Proactive SC`, `Autonomic SC`, `Parallel AMs SC`, `Multilevel AMs SC`, `Centralized SCE`, `P2P AMs SCE`, `Cognitive Stigmergy SCE`, `Hierarchy of AMs SCE patterns`). The

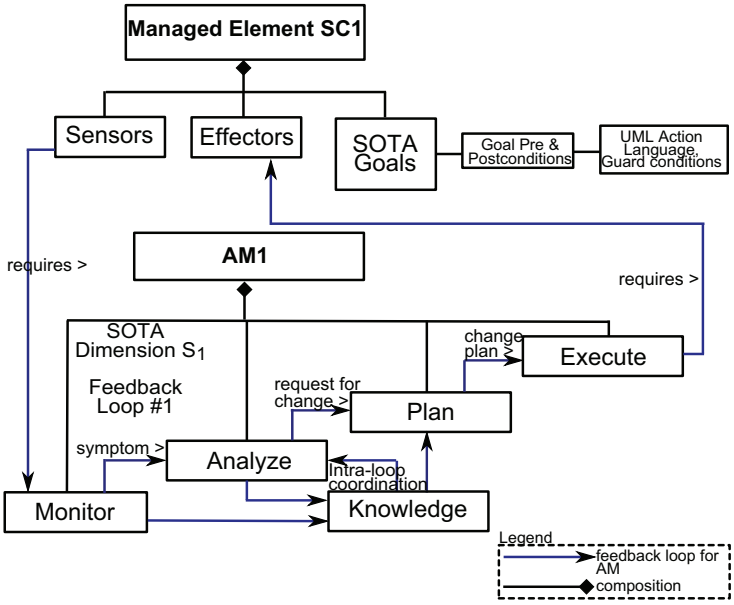


FIG. 3.1. *Autonomic SC pattern: Conceptual model.*

main goal behind this is to facilitate general-purpose and application-independent instantiation of models for complex systems based on feedback loops. More specifically, it [1]:

(i) provides the engineer a starting structure for pattern modelling activities in support of capturing details related to patterns modelling. The templates are used to specify any architecturally significant structures that need to be included in the activity-based pattern models created using the templates.

(ii) presents useful guidance and textual advice to the engineer on applying the profile and deriving platform-specific models in a consistent manner. It can provide instructions to the engineer on how to fill and complete the model using elements within the template, and using features of the tool environment.

For a high-level and conceptual description of the key SOTA patterns of the catalogue refer to [13]. At the SC level, these patterns use a decentralized feedback loop approach while at the SCE level, they primarily use a centralized feedback loop approach. Thus, our work applies both decentralized and centralized feedback loop techniques in order to exploit their benefits. Although these template models are not Eclipse plug-ins, they are distributed in *plug-ins*. Here, we present UML template models created for three key SOTA architectural patterns at the SC (`Autonomic SC pattern`, `Parallel AMs SC pattern`) and SCE (`Centralized SCE pattern`) levels. The pattern templates have been modelled using UML 2.2 activity models (cf. Figs. 3.2, 3.4 and 3.6). The corresponding conceptual models of these patterns are provided in cf. Figs. 3.1, 3.3 and 3.5.

**3.2.1. Autonomic SC Pattern.** The `Autonomic SC pattern` is characterized by the presence of an explicit, external feedback loop to direct the behaviour of the managed element (cf. Figs. 3.1 and 3.2). The managed element has sensors, effectors and a representation of SOTA goals. The SOTA utilities are enforced in the managers. An *autonomic manager (AM)* handles the adaptation of the managed element.

In general, an SC and its manager model the following feedback loop behaviour [1]. The sensors in the SC capture event sensor data. This is then collected by the monitor phase of the manager, which filters and accumulates the event data. The event data is stored in the knowledge base component of the loop, which also stores predicate rules for the SOTA utilities. Then, the analyse phase of the loop gathers the event signals and utilities from the knowledge base component of the loop and interprets against the patterns. The result of this interpretation (event symptoms) is stored in the knowledge base component of the loop. The plan phase of the loop obtains event symptoms and interprets them. If the awareness level is not satisfied then it triggers and devises a plan to execute awareness change. To achieve this, the execute phase of the loop notifies the SC effector which adapts the required awareness level inside the SC accordingly.

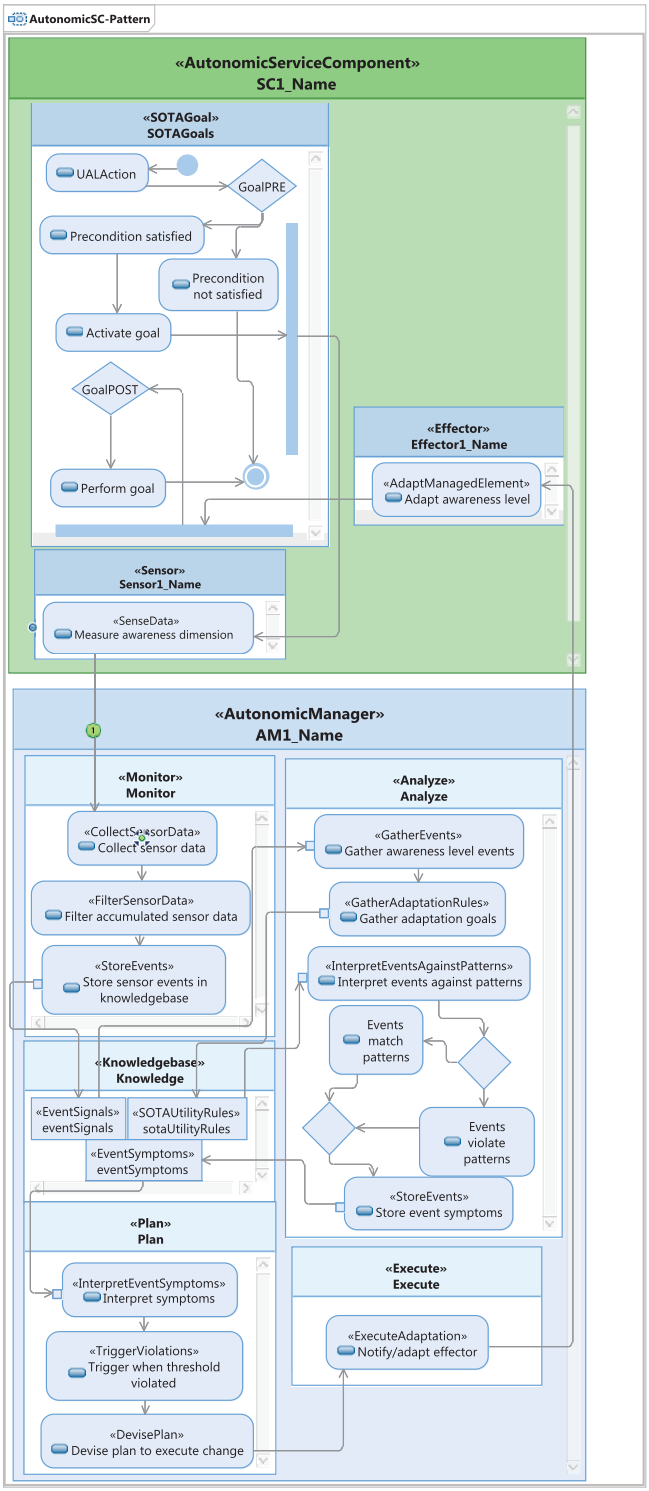


FIG. 3.2. *Autonomic SC pattern (continued): UML pattern template model.*

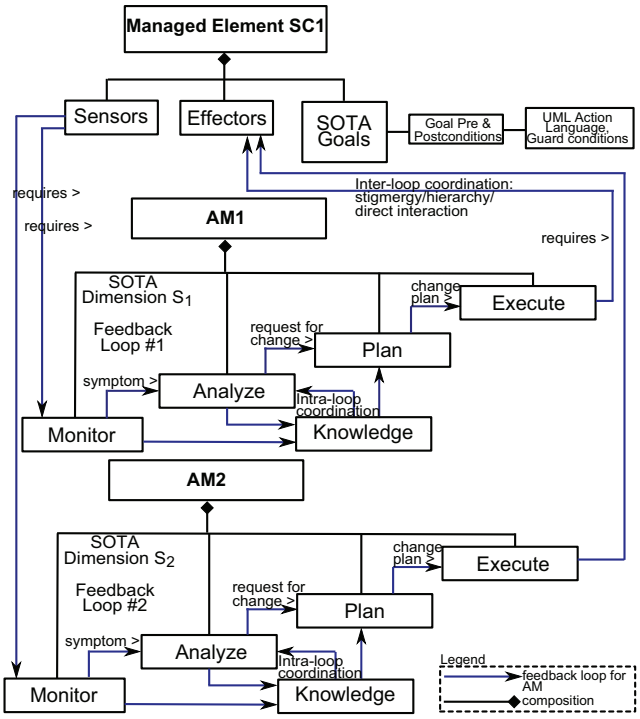


FIG. 3.3. *Parallel AMs pattern: Conceptual model.*

**3.2.2. Parallel AMs SC Pattern.** The `Parallel AMs SC pattern` is an extension of the `Autonomic SC pattern` [13]. Here, several AMs can be associated with the managed element, each closing a feedback loop devoted to controlling a specific adaptation aspect of the system (cf. Figs. 3.3 and 3.4). Adding different levels of AMs increases the autonomicity, and these extra AMs work in parallel to manage the adaptation of the managed element. For instance, let us consider that we have two feedback loops with an AM in each loop to handle adaptation in two SOTA dimensions. These loops can interact with each other using *hierarchy*, *stigmergy* or *direct interaction*, and here we can identify an *inter-loop* coordination where MAPE-K computations of the loops coordinate with each other. Also, an *intra-loop* can be identified between the Analyse and Knowledge components of an AM to allow the coordination of adaptation between these two phases.

**3.2.3. Centralized SCE Pattern.** The `Centralized SCE pattern` is characterized by a global feedback loop, which manages a higher-level adaptation of behaviour of multiple autonomic components (cf. Figs. 3.5 and 3.6) [1]. The adaptation in the `Centralized SCE pattern` is handled by a *super AM* which is a high-level AM. Like an AM, a super AM has the MAPE-K adaptation model. This is while the single SCs are able to self-adapt their individual behaviour using their own external feedback loops. The feedback loops in this pattern can interact using *hierarchy*, *stigmergy* or *direct interaction*.

**3.3. Java Templates for the Patterns.** We provide a set of domain-independent templates in Java for the key SOTA patterns [1]. These domain-independent templates can be used as examples by the developer when implementing the patterns in their domains. An alternative solution can be to provide a reusable library for the key SOTA patterns. However, it is impractical to generalize programming patterns such that they could be reused across different complex systems based on feedback loops.

We have used the *Fork/Join framework* of Java SE 7 to implement the SOTA patterns that have `autonomic SCs` as the constituent SCs. The implementation of SOTA patterns that have other types of SCs as constituent SCs (e.g. `primitive SCs` and `proactive SCs`) is currently work in progress. In SOTA, *goals* represent the eventual state of the affairs that a system or component has to achieve [2]. On the other hand, *utilities* are constraints on the trajectory or execution path that a system should try to respect while achieving the goals.

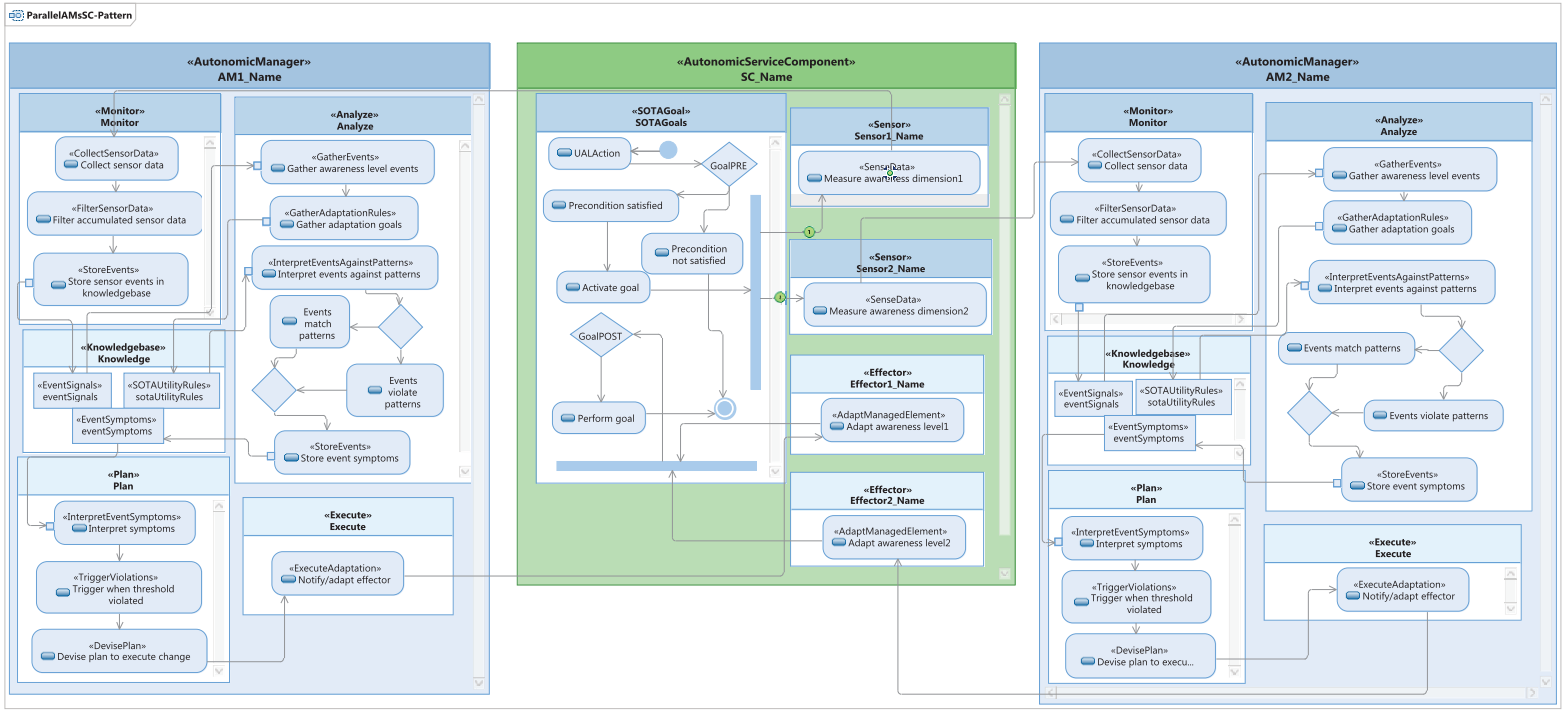


FIG. 3.4. *Parallel AMs pattern (continued): UML pattern template model.*

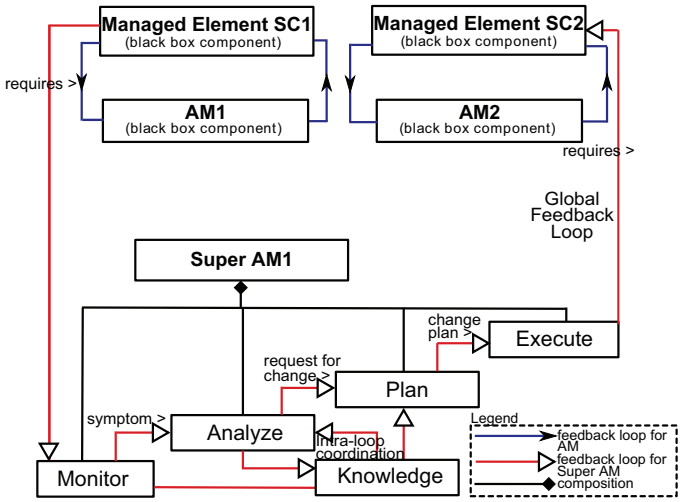


FIG. 3.5. *Centralized SCE pattern: Conceptual model.*

In the SOTA patterns that have `autonomic SCs` as the constituent SCs, the goals are performed by the SCs while utilities are handled by their respective autonomic managers. This design principle maps and corresponds well to the *Fork/Join Framework* of the Java SE 7, which has introduced the notion of *parallelism*. Using the Fork/Join framework, an SC delegates its awareness and adaptation handling activities (utilities) to its AMs. Utilities are enforced parallel to the execution of an entity in the SOTA space.

Each Java template (e.g. cf. Figs. 3.7 and 3.8) maps well to their corresponding UML design templates discussed previously. In all the templates, in general, an AM or a super AM has methods to handle the monitor, analyse, plan and execute phases of the feedback loops (e.g., `monitor()`, `analyse()`, `plan()`, `execute()` methods in cf. Fig. 3.7). HashMaps have been created to deal with the knowledge base elements of the loops (e.g. a HashMap called `am1EventSignals` to store event signals in the Knowledge base). On the other hand, in the SCs, the preconditions and the postconditions of the SOTA goals have been implemented as predicate rules. Also, an SC has methods to handle the behaviour of its sensors and effectors. As for the interactions, the AMs can implement behaviour for *intra-loops*, for example, the analyse method handling the analyse phase of the loop queries and stores data in the HashMaps, simulating an intra-loop behaviour. Meanwhile, the behaviour for *inter-loops* (i.e. *stigmergy*, *hierarchy* and *direct interaction*) have been implemented as method calls. Stigmergy has been implemented implicitly where two AMs communicate with each other through the SC they are connected with (e.g. `Parallel AMs SC pattern`).

For example, the Java template for the `Parallel AMs SC pattern` (cf. Fig. 3.7) has two AM classes associated with a single SC class, each closing a feedback loop controlling a specific adaptation aspect of the system. On the other hand, the template provided for the `Centralized SCE pattern` (cf. Fig. 3.8) has two SC classes and a super AM class. Out of these patterns, the `Autonomic SC` and `Parallel AMs SC` patterns exhibit a decentralized feedback loop approach, while the `Centralized SCE` pattern demonstrates a centralized feedback loop approach.

**4. Case Study Scenario.** This section describes the e-mobility case study problem used to derive platform-specific models in the pattern engineering and implementation process. The case study problem addresses the SOTA model's self-awareness and self-adaptation mechanisms.

The case study scenario concerns individual planning and mobility for a single user and a privately owned vehicle. Let us consider a situation where a user intends to travel to an appointment at a particular destination (cf. Fig. 4.1, top) [9]. First, the user drives the electric vehicle (e-vehicle) to the car park, then parks the car and walks to the meeting location. During the walking and meeting times, the e-vehicle can be recharged. Here, the main SCs are the user or driver, the e-vehicle, and the parking lots and charging stations (infrastructure). A main SCE in this scenarios is the temporal orchestration of the user, the e-vehicle, a parking lot and a charging

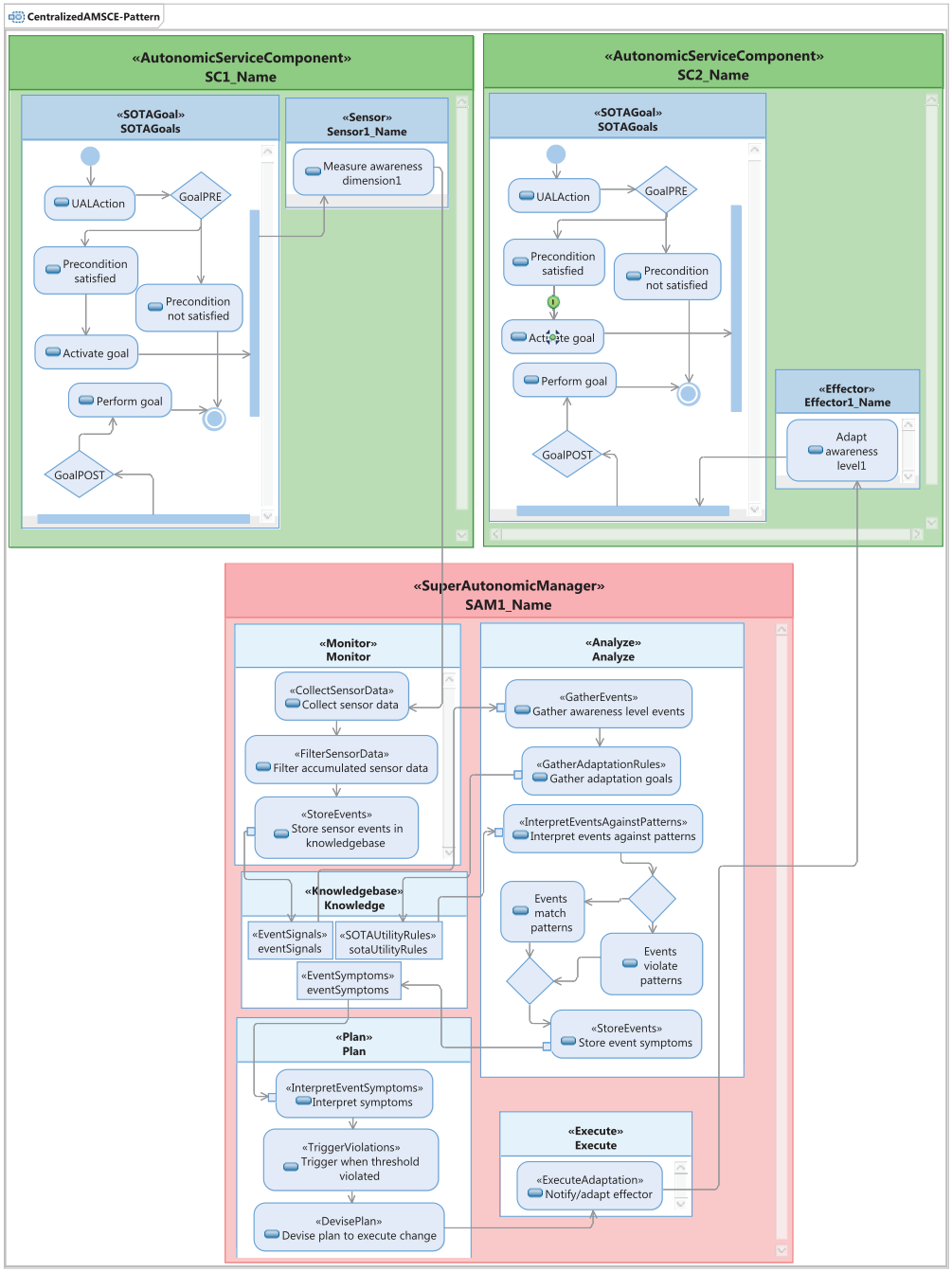


FIG. 3.6. *Centralized SCE pattern (continued): UML pattern template model.*

station assigned at trip start (*User-E-Vehicle-Assigned-Infrastructure SCE*, cf. Fig. 4.1).

The e-mobility case study problem is well suited to be solved using appropriate self-awareness and self-adaptation mechanisms (e.g. the SOTA model) for the following reasons [4]:

(i) The case study problem involves a significant number of SCs. Therefore, relying only on centralized solutions becomes non-feasible and providing for decentralized solutions to handle localized decision making is

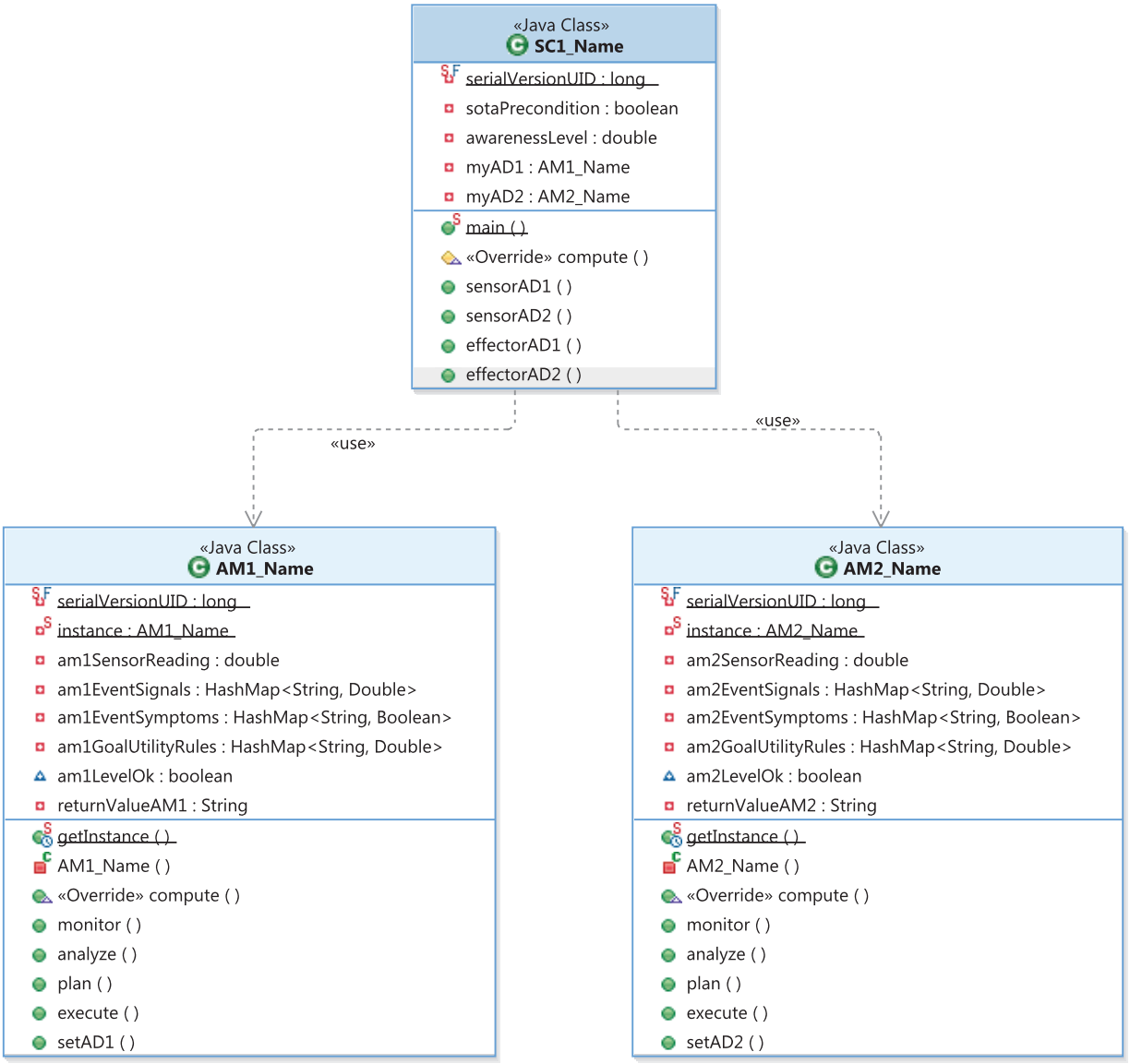


FIG. 3.7. *Java pattern template models: Parallel AMs SC pattern.*

important.

(ii) This problem deals with several awareness dimensions that need to be handled by the complex SCs, e.g. e-vehicle SC: time, energy, location; user SC: user preferences (climate comfort, driving style), time, cost; parking lots and charging stations SCs: availability.

(iii) Each SC has access only to partial information or partial view of the environment (e.g. during the trip, the e-vehicle is not aware of the availability of its assigned parking lot). Therefore, the individual SCs need strategies to adapt at run-time as more information becomes available to them.

(iv) Each SC and SCE is involved in significant levels of uncertainty or contingency situations (system or environmental changes) requiring self-adaptive actions. These can be (1) e-vehicle SC: the unavailability of a parking lot; the planned event deadline is missed (the e-vehicle could not reach the destination at the time required or with the energy planned); the user overrides the plan; (2) user SC: the shifting of an appointment;


FIG. 3.8. *Java pattern template models (continued): Centralized SCE pattern.*

and (3) parking lot and charging station SCs: the e-vehicle does not arrive at the booked time or it leaves earlier or later; charging is not initiated at the foreseen time and draws unforeseen amounts of power.

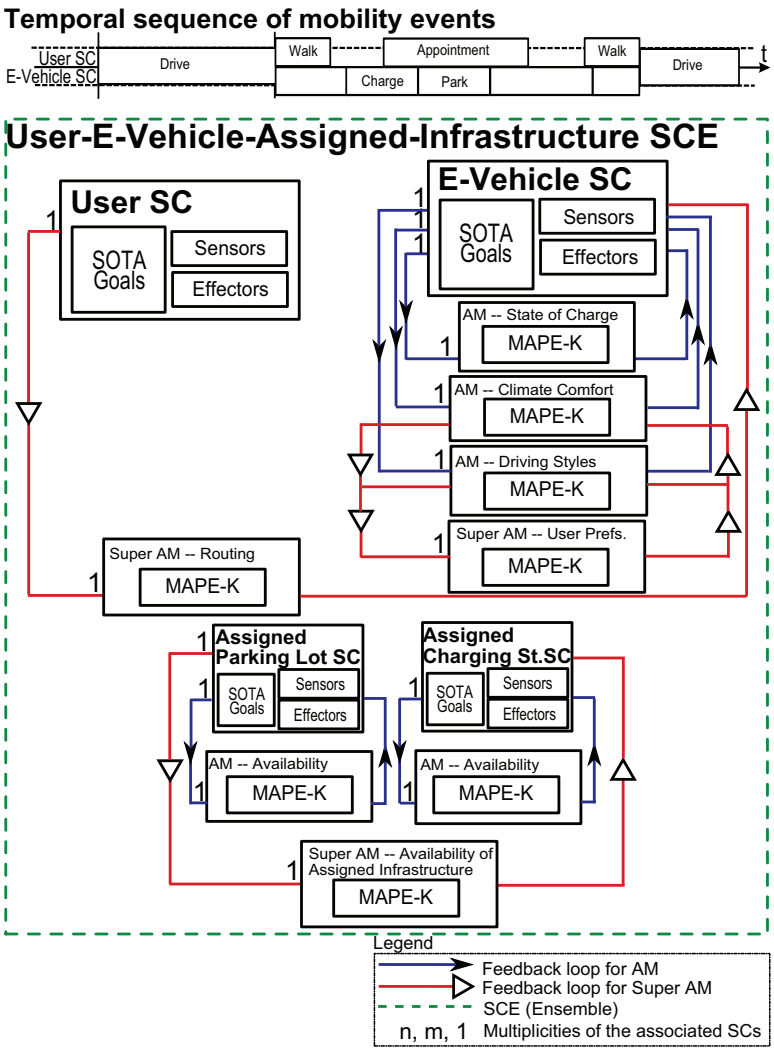


FIG. 4.1. *SOTA patterns applied to the e-mobility scenario.*

**4.1. Case Study Scenario and Automation with SOTA.** In the SOTA model, the SCs are conceptually modelled as entities moving and executing in the space, and these entities can have goals and utilities that describe how such goals can be achieved at an individual or a global level. Each SC and SCE of the mobility scenario can be described using the SOTA goals and utilities, the awareness being monitored for each managed element, and the self-adaptive behaviour using SOTA feedback loops as a response to any contingencies that may occur.

For example, let us consider the e-vehicle SC, which is the central SC within the mobility scenario. It interfaces with both the user and the infrastructure SCs during driving, and with the infrastructure SCs only during parking or walking. The goal of the e-vehicle is to reach the destination with the planned energy and at the planned time. A utility can be the constraint that the battery charge should not reach *low* until the e-vehicle reaches its destination. The monitored awareness dimensions are, among others, the state of the battery charge, temperature (for climate comfort requirements of the user), acceleration and velocity (for driving style requirements), and current location of the vehicle.

As shown in cf. Fig. 4.1, in order to handle the adaptation of a managed element, we can provide separate AMs (`Autonomic SC pattern` - Sect. 3.2.1) for each SOTA awareness dimension. The e-vehicle SC has three

AMs defined to handle the adaptation of the battery state of charge, climate comfort and driving style requirements of the user. Such separate AMs here provides an instance of the `Parallel AMs SC pattern` described in Sect. 3.2.2. Also, any self-adaptive behaviour on routing needs to be handled at the SCE level, as these actions are applicable to both the user and the e-vehicle SCs. Thus, a super AM has been defined to handle the adaptation of routing (`Centralized SCE pattern`). The parking lot SCs and charging station SCs need to be aware of their availability. To handle possible contingency situations, two separate AMs (`Autonomic SC pattern`) can be defined to manage the availability of the two assigned infrastructure SCs, and a super AM can be provided to handle the adaptation of both SCs (`Centralized SCE pattern`).

As seen here, it is clear that there are a number of SCs, SCEs, AMs and super AMs closing multiple, interacting feedback loops. These feedback loops, which SOTA uses as mechanisms to express self-awareness and self-adaptation, can be organized using several architectural patterns. Therefore, a key goal of our work is to provide engineering and implementation support to the software engineer in order to easily grasp this complex setup.

**5. Domain-Specific Pattern Models.** The domain-specific pattern models created at the platform-independent UML and platform-specific Java levels for the case study scenario are discussed in this section. Model transformations have been employed as a bridge to enforce correct separation of concerns between these two design abstraction levels.

**5.1. Platform-Independent Pattern Models.** We have used the simulation plug-in component of the integrated SimSOTA tool to instantiate the UML pattern template models for the e-mobility case study. UML 2.2 activity models have been used as the primary notation to model the behaviour of feedback loops. In a feedback loop, the actions are not necessarily performed sequentially. An iterative process allows the revision of certain decisions if required, and therefore activity diagrams are effective to design the feedback loops. The plug-in also facilitates the simulation of feedback loops in other UML 2.2 diagrams, such as composite structure and sequence diagram models [4].

We now briefly describe the domain specific, platform-independent models realized using SimSOTA to simulate a number of feedback loop structures in the *User-E-Vehicle-Assigned-Infrastructure SCE* of the e-mobility scenario (cf. Fig. 4.1). Note that due to space considerations, the activity model provided in cf. Fig. 5.1 is a subset of the *User-E-Vehicle-Assigned-Infrastructure SCE*. Here, we particularly describe the instantiation of three key SOTA patterns - `Autonomic SC pattern`, `Parallel AMs SC pattern`, and `Centralized SCE pattern`.

There are several managed elements or SCs: the e-vehicle (`SC_EVehicle`; cf. Fig. 5.1, top-center), the user (`SC_User`) and the assigned parking lot (`SC_ParkingLotAssigned`). Each managed element (e.g. e-vehicle SC) has an activity-based UML model to represent SOTA goals (e.g. reach destination) which can be characterized in terms of a precondition (e.g. whether the assigned parking lot is available) and a postcondition (e.g. actual reaching of the destination within the state of battery charge and time). The preconditions and postconditions of the SOTA goals are modelled using UML Action Language and guard conditions. The utilities for the e-vehicle SC are constraints on the state of the battery charge, climate comfort, driving style (acceleration and velocity), and routing requirements until the e-vehicle reaches its destination. The utilities are modelled in the managers.

**5.1.1. Decentralized and Centralized Feedback Loops and Interactions.** The SimSOTA simulation plug-in integrates both decentralized and centralized feedback control loop techniques to handle the adaptation of the managed elements. In this example, the decentralized feedback loop behaviour is provided by the `Autonomic SC pattern` and `Parallel AMs SC pattern`. Here, the `SC_EVehicle` and the AM on battery charge (`AM_EV_SoC`) form an instance of the `Autonomic SC pattern`. The `Sensor` components in `SC_EVehicle` form concurrent activities for measuring the battery charge, temperature (for climate comfort), location (for routing), acceleration and velocity (for driving style) inside the e-vehicle, respectively (cf. Fig. 5.1, top-center). These concurrent activities close several decentralized feedback loops in the AMs which interact using stigmergy and act on its shared subsystem. On the other hand, the `SC_EVehicle` and the two AMs on battery charge (`AM_EV_SoC`) and climate comfort (`AM_EV_ClimateComfort`) form an instance of the `Parallel AMs SC pattern` (cf. Fig. 5.1).

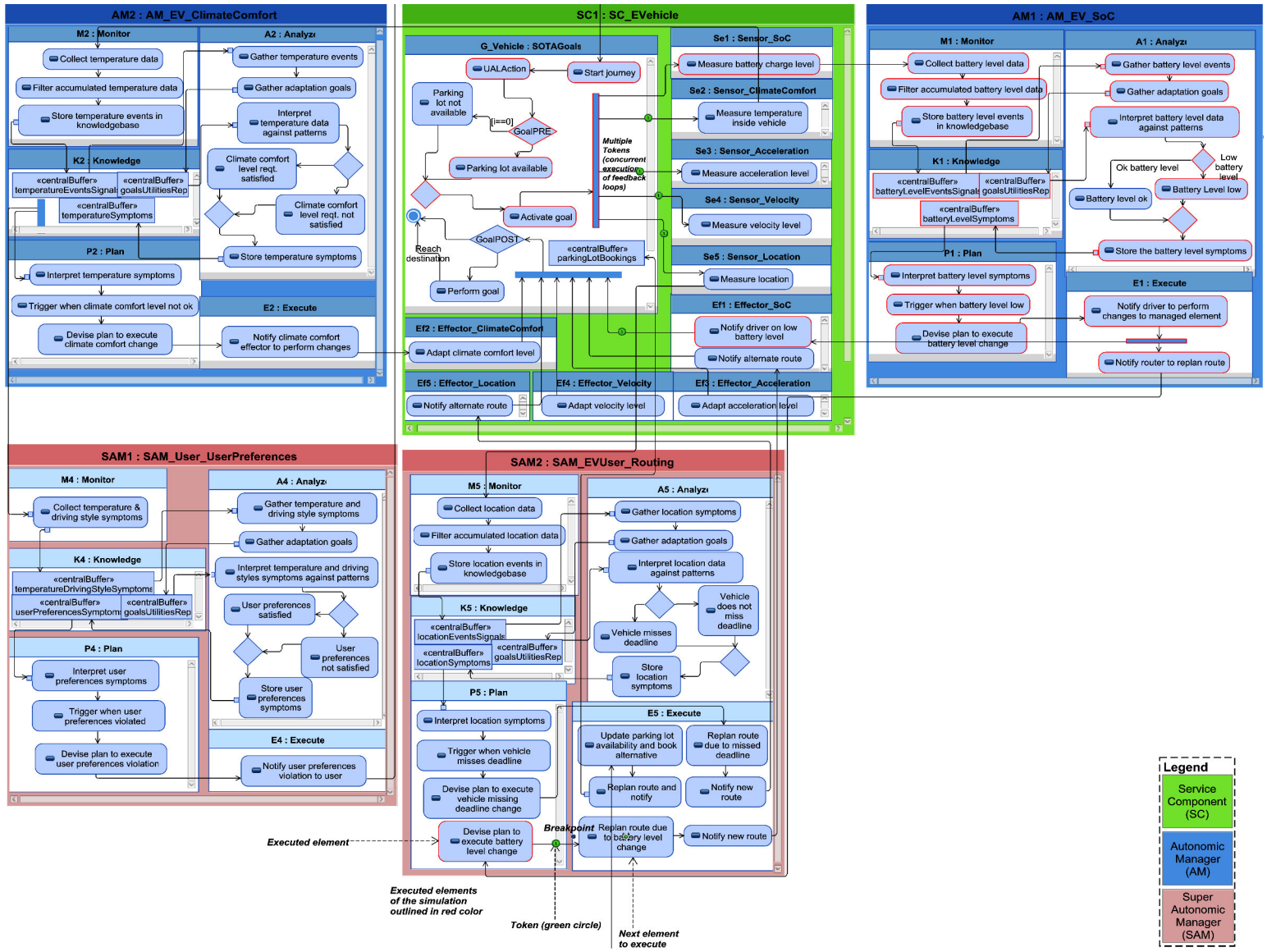


FIG. 5.1. *Patterns simulated as an activity model for an e-mobility scenario.*

The super AM defined for managing the adaptation of routing (`SAM_EVUser_Routing`; cf. Fig. 5.1, bottom-center) closes a separate, centralized (global) feedback loop. The super AM deals with both the user and the e-vehicle SCs. This instantiates the `Centralized SCE pattern`. The `SAM_EVUser_Routing` deals with contingency situations if the e-vehicle misses its deadline. Refer to [4] for more details on the domain-specific, platform-independent models (and interactions) derived in the example.

**5.2. Transformations.** Model transformations have been used in the current study to automate the generation of Java-based implementation code for the complex patterns [1]. We have initially applied model-to-text Java Emitter Template (JET) transformation to transform the UML activity model-based SOTA patterns into textual Java. JET is an open-source technology developed by IBM and are typically used in the implementation of a code generator [7]. Here, the transformation contains XPath expressions to navigate the UML activity models created for the patterns and extract model information dynamically to the transformation. However, as JET's support for UML models has several limitations, we have used a more effective method to transform the UML activity diagram-based SOTA patterns into textual Java.

This multi-stage transformation chain describes an effective pipeline of model-to-model and model-to-text JET transformations. In this solution, first a model-to-model mapping transformation was created which extracts relevant information from the UML activity model elements and stereotypes, and then a code generator specific EMF intermediate model was built which contains only information required for the back-end model-to-text JET transformation. The front-end model-to-model transformation automatically invokes the back-end JET transformation.

**5.3. Platform-Specific Pattern Models.** The transformations generated Java files for the SOTA patterns in e-mobility can be elaborated by the engineer to derive a complete implementation for the patterns. For this, the engineer can use as examples the set of domain-independent Java templates discussed in Sect. 3.3. In this study, we have instantiated these templates to implement the behaviour of the SCs and managers involved in the e-mobility case study scenario. To this end, we have implemented several self-adaptation scenarios. For example, (1) the e-vehicle's energy level is inadequate to follow the plan. This is managed by the AM class defined for state of charge on the vehicle SC class; (2) the e-vehicle's climate comfort level is not satisfied, which is handled by the AM class specified to handle climate comfort for the vehicle; (3) the driving style requirements (velocity, acceleration) are not satisfied by the e-vehicle for the user. This is managed by the AM class defined for driving style; (4) the e-vehicle has missed (or is going to miss) a deadline of a planned event. This is managed by the super AM class defined for routing; (5) the user preferences are not satisfied. This is handled by the super AM for user preferences; (6) the e-vehicle does not arrive at the booked time. This is handled by the AM for parking lot availability; (7) the e-vehicle leaves earlier/later than booked time. This is handled by the AM class defined for parking lot availability.

In this manner, we have implemented and validated the SOTA patterns in the e-mobility case study. This shows that the patterns can be simulated and implemented effectively and that they can be effectively applied to a case study.

**5.3.1. SimSOTA Installation and Usage.** The distribution scheme that will be adopted for SimSOTA relies on the Eclipse platform feature export. The entire SimSOTA tool can then be downloaded using the standard Eclipse update site mechanism. An update site is a mechanism for finding and installing features. In order to export the plug-ins in the SimSOTA tool, first, a feature project that references the plug-ins will be created. Second, an update site will be created to distribute the feature created. Finally, the Eclipse Update Manager can be used to scan update sites for the newly created feature for SimSOTA and install it. The installed plug-ins can be executed in the IBM Rational Software Architect simulation environment. At this moment, however, a packaged version of SimSOTA is not publicly available, due to its dependencies on the non-free IBM Rational Software Architect simulation environment.

**6. Discussion and Related Work.** Our approach and SimSOTA plug-in offers several benefits in the domain of self-adaptive architectural patterns engineering and implementation. SimSOTA is an integrated eclipse plug-in with several plug-ins grouped together, i.e. a simulation plug-in and transformation plug-ins. The idea is to provide an integrated environment for the architect to engineer and implement SOTA patterns in their respective domains. A key goal of SimSOTA simulation plug-in is to facilitate the engineering (modelling,

simulating and validating) of complex self-adaptive systems based on feedback loops. This is to facilitate general-purpose and application-independent instantiation of models for complex systems based on feedback loops. To achieve this, as described in the paper, we have provided a set of pattern templates (UML and Java) for all the key SOTA patterns in the catalogue. The SimSOTA plug-in can also be used to simulate and animate the patterns to better understand their complex and dynamic model behaviour. Also, the feedback loop models and their interplay can be validated to detect errors early and to check whether the specified behaviour works as intended. The transformation plug-ins of the integrated SimSOTA tool facilitate the automatic generation of implementation code in Java for the patterns. The use of transformations is beneficial here as it acts as a bridge to enforce correct separation of concerns between two design abstractions, i.e. UML-based patterns and their Java implementations. Other benefits include improving the quality and productivity of patterns development, and easing system maintenance and evolution for the patterns engineer.

Furthermore, our model-driven engineering approach integrates both decentralized and centralized feedback loop techniques in order to exploit their benefits. Centralized approaches allow global behaviour control, but they contain a single point of failure and suffer from scalability issues. Meanwhile, decentralized approaches do not require any a priori knowledge, nor do they contain a single point of failure. The integration of these approaches was evident with the use of a collection of architectural patterns at both SC (decentralized - e.g. `Autonomic SC pattern` and `Parallel AMs SC pattern`) and SCE (centralized - e.g. `Centralized SCE pattern`) levels.

There are several works in the literature that are related to the current research as presented next. Several authors (e.g. [12, 8, 15, 14, 16, 11, 20]) have emphasized the need to make feedback loops first-class entities in self-adaptive systems. Muller, Pezze and Shaw [12] discuss an approach to increase the visibility of control loops to support their continuous evolution. They highlight the need for multiple control loops in an adaptive ultra large-scale system, and stress the need for refining the loops into reference models and design patterns. Although these ideas have been discussed at the conceptual level, no implementation or validation of the work using techniques such as simulations has been reported [12].

Vromant et al. [15] describe an implementation framework that extends IBM's MAPE-K model with support for two types of adaptation coordination: intra-loop and inter-loop. While their work is comprehensive in coordinating and integrating multiple control loops, the MAPE-K loops used do not support an integration of centralized and decentralized adaptation coordination as provided in our work.

To manage the complexity of internet applications, the authors in [14] propose a set of weakly interacting (i.e. stigmergy, hierarchy and direct interaction) feedback structures where each structure consists of a set of feedback loops to maintain one system property. As in our work, in [16] a feedback loop has been represented as a flow of information and actions, and UML activity diagrams have been used as notation. However, unlike our approach, both [14] and [16] have not provided detailed individual steps of the adaptation process nor have they provided tool support for modelling, simulation and validation of the feedback loop models.

Vogel and Giese [20] establish a domain-specific modelling language for run-time models called *megamodels* and an interpreter to execute them. Using the modelling language, single and multiple feedback loops and their interplay can be explicitly specified in the megamodels. Like our approach, their work has considered detailed individual adaptation steps like monitoring, analysis, planning and execution. Also, their modelling language is similar to the UML activities used in our work with respect to modelling flows of actions or operations. However, the authors have not considered any validation of the simulated feedback loop models in [20].

Compared to previous approaches to the engineering and implementation of self-adaptive systems using feedback loops, to the best of our knowledge, there is very little implementation or tool support to address the needs of software architects. The present work aims to contribute to this end.

**7. Conclusions and Future Work.** In this paper, we presented SimSOTA, which is an integrated Eclipse plug-in tool we have developed to engineer and implement self-adaptive systems based on our feedback loop-based approach. SimSOTA facilitates both modelling and simulating of complex patterns, and the generation of Java-based implementation code for the patterns using transformations. The approach integrates both decentralized and centralized feedback loop techniques in order to exploit their associated benefits. The approach and plug-in have been validated and assessed using a case study in cooperative electric vehicles.

As part of our future work, we will exploit the results of our simulation and implementation experiences to produce effective software engineering guidelines to facilitate the development of self-adaptive application with

SimSOTA. Second, we plan to integrate the SOTA patterns defined at the conceptual, UML and Java levels with the rest of the e-mobility framework for further validation. Finally, we plan to assess SimSOTA in the context of different application areas, namely swarm robotics systems and distributed cloud systems.

**Acknowledgments.** This work is supported by the ASCENS project (EU FP7-FET, Contract No.257414).

REFERENCES

[1] D. B. Abeywickrama, N. Hoch, and F. Zambonelli, *An integrated Eclipse plug-in for engineering and implementing self-adaptive systems*, Proceedings of the IEEE 23rd International WETICE Conference, IEEE (2014), pp. 3–8.

[2] D. B. Abeywickrama, N. Bicocchi, and F. Zambonelli, *SOTA: Towards a general model for self-adaptive systems*, Proceedings of the IEEE 21st International WETICE Conference, IEEE (2012), pp. 48–53.

[3] D. B. Abeywickrama, and F. Zambonelli, *Model checking goal-oriented requirements for self-adaptive systems*, Proceedings of the IEEE 19th International Conference and Workshops on Engineering of Computer Based Systems (ECBS), IEEE (2012), pp. 33–42.

[4] D. B. Abeywickrama, N. Hoch, and F. Zambonelli, *SimSOTA: Engineering and simulating feedback loops for self-adaptive systems*, Proceedings of the 6th International C* Conference on Computer Science & Software Engineering (C3S2E'13), ACM (2013), pp. 139–144.

[5] D. B. Abeywickrama, F. Zambonelli, and N. Hoch, *Towards simulating architectural patterns for self-aware and self-adaptive systems*, Proceedings of the 2nd Awareness Workshop co-located with the SASO'12 Conference, IEEE (2012), pp. 133–138.

[6] G. Cabri, M. Puviani, and F. Zambonelli, *Towards a taxonomy of adaptive agent-based collaboration patterns for autonomic service ensembles*, Proceedings of the International Conference on Collaboration Technologies and Systems, IEEE (2011), pp. 508–515.

[7] J. DeCarlo, L. Ackerman, P. Elder, C. Busch, A. Lopez-Mancisidor, J. Kimura, and R. S. Balaji, Strategic Reuse with Asset-Based Development, IBM Corporation, Riverton, New Jersey, USA (2008).

[8] R. Hebig, H. Giese, and B. Becker, *Making control loops explicit when architecting self-adaptive systems*, Proceedings of the 2nd International Workshop on Self-Organizing Architectures, ACM (2010), pp. 21–28.

[9] N. Hoch, K. Zemmer, B. Werther, and R. Y. Siegwart, *Electric vehicle travel optimization—customer satisfaction despite resource constraints*, Proceedings of the 4th Intelligent Vehicles Symposium, IEEE (2012), pp. 172–177.

[10] J. O. Kephart and D. M. Chess, *The vision of autonomic computing*, IEEE Computer (2003), 36(1):41–50.

[11] M. Luckey, B. Nagel, C. Gerth, and G. Engels, *Adapt cases: extending use cases for adaptive systems*, Proceedings of the 6th International SEAMS Symposium, ACM (2011), pp. 30–39.

[12] H. Müller, M. Pezzè, and M. Shaw, *Visibility of control in adaptive systems*, Proceedings of the 2nd International Workshop on Ultra-large-scale Software-intensive Systems, ACM (2008), pp. 23–26.

[13] M. Puviani, G. Cabri, and F. Zambonelli, *A taxonomy of architectural patterns for self-adaptive systems*, Proceedings of the 6th International C* Conference on Computer Science & Software Engineering (C3S2E'13), ACM (2013), pp. 77–85.

[14] P. Van Roy, S. Haridi, and A. Reinefeld, *Designing robust and adaptive distributed systems with weakly interacting feedback structures*, Technical report, ICTEAM Institute, Universit catholique de Louvain (2011).

[15] P. Vromant, D. Weyns, S. Malek, and J. Andersson, *On interacting control loops in self-adaptive systems*, Proceedings of the 6th International SEAMS Symposium, ACM (2011), pp. 202–207.

[16] T. De Wolf and T. Holvoet, *Using UML 2 activity diagrams to design information flows and feedback-loops in self-organising emergent systems*, In T. De Wolf, F. Saffre, and R. Anthony, editors, Proceedings of the 2nd International Workshop on Engineering Emergence in Decentralised Autonomic Systems (2007), pp. 52–61.

[17] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, et al, *Software engineering for self-adaptive systems: A research roadmap*, Software Engineering for Self-Adaptive Systems, volume 5525 of *LNCS*, Springer-Verlag (2009), pp. 1–26.

[18] E. Clayberg and D. Rubel, *Eclipse Plug-ins*, Addison-Wesley Professional, 3 edition, 2008.

[19] T. Haupt, *Towards mediation-based self-healing of data-driven business processes*, Proceedings of the 7th International SEAMS Symposium, IEEE/ACM (2012), pp. 139–144.

[20] T. Vogel and H. Giese, *A language for feedback loops in self-adaptive systems: Executable runtime megamodels*, Proceedings of the 7th International SEAMS Symposium, IEEE/ACM (2012), pp. 129–138.

[21] J. Wuttke, Y. Brun, A. Gorla, and J. Ramaswamy, *Traffic routing for evaluating self-adaptation*, Proceedings of the 7th International SEAMS Symposium, IEEE/ACM (2012), pp. 27–32.

*Edited by:* Giacomo Cabri
*Received:* September 15, 2014
*Accepted:* January 5, 2015