Engineering Self-Adaptive Systems with the Role-Based Architecture of HELENA

Annabelle Klarl
Ludwig-Maximilians-Universität München, Germany

***Abstract*—When engineering self-adaptive systems, separating adaptation and application logic was proven beneficial to avoid interdependencies between adaptation strategy and standard behavior. Several engineering methods support this separation in different phases of the classical development process, but none addresses it consistently in all of them. We propose a holistic model-driven engineering process with systematic transitions between all phases to develop self-adaptive systems. Adaptation is achieved by changing the behavioral mode of a component in response to perceptions. We realize behavioral modes by roles which a component can dynamically adopt. For specification, we propose adaptation automata which allow to specify complex adaptation behavior by hierarchical structure and history of states. Furthermore, we propose the HELENA Adaptation Manager pattern to derive a role-based model from a specification. Due to its formal foundation, the model can be analyzed with Spin and executed with the Java framework jHELENA.**

I. INTRODUCTION

To cope with changing conditions at run-time, the concept of self-adaptation has been proposed. A self-adaptive component keeps track of its individual and shared goals, perceives its internal state as well as its environment, and adapts its behavior accordingly [1], [2]. We say, a self-adaptive component changes its *behavioral mode* in response to perceptions.

Engineering self-adaptive systems (state of the art): It is commonly agreed that adaptation logic of self-adaptive systems (SAS) should be developed independently from application logic [1], [3], [4]. Thus, the application logic of behavioral modes is developed without considering changes in the environment. Conversely, adaptation logic is designed to switch between behavioral modes without taking care how the modes perform their task. In the literature of SAS, this separation of concerns is addressed at different development phases: Automata-based approaches [5]–[8] offer formal specification and verification techniques. Architectural patterns [9] introduce design guidelines for realization. Role models [10], [11] propose concepts for switching between behavioral modes. Architecture-based self-adaptation [12]–[14] is presented as a framework for designing and implementing SAS.

Consequences: None of the existing approaches provides a holistic development process considering adaptation logic independently from application logic in all main phases and supporting systematic transitions between all of them. Therefore, artifacts cannot be easily traced through the whole process and we cannot guarantee correct realization of requirements since we lack systematic transitions between each of the phases.

This work was partially sponsored by the EU project ASCENS, 257414.

Contributions: Integrating and extending existing approaches, we propose a holistic model-driven engineering process to develop self-adaptive systems. (1) For specification, we propose *hierarchical adaptation automata*. They offer history states and hierarchical composition of states as expressive tools to specify complex adaptation behavior. Furthermore, they provide placeholders to plug application logic in. (2) For the design, we propose a role-based architecture following the HELENA *Adaptation Manager (HAM) pattern*. Roles intuitively express the different tasks of self-adaptive components, like being aware of the environment, managing adaptation, and performing particular behavioral modes. (3) We propose a *systematic transition* (which can be fully automated) from specification to role-based design. Especially, representing an adaptation automaton as a standard labeled transition automaton is particularly involved since hierarchical structure and history states have to be resolved. (4) By realizing the role-based design with the modeling approach HELENA [15], we benefit from its *implementation and verification tools* through reusing its automatic transformations to Java [16], [17] and Promela [18]. (5) Thus, we introduce the missing *traceability of artifacts* throughout the whole engineering process while keeping adaptation logic and application logic separated.

Outline: The HELENA development process is shown in Fig. 1. In this paper, we focus on the first two phases and illustrate them with a search-and-rescue scenario (cf. Sec. II)

(1) Adaptation specification (Sec. III): The self-adaptive components which contribute to the system are specified by their *signature* – capturing attributes and behavioral modes of the component – and an *adaptation automaton* – describing when the component switches between behavioral modes.

(2) Design pattern (Sec. IV): We propose the *HAM pattern* to create a role-based model for self-adaptive components. We exploit the concept of roles to encapsulate modes and their behaviors. A component's behavior is adapted by changing the currently active role of the component. An adaptation manager, also represented by a role, takes care to switch between mode roles according to the adaptation automaton.

(3) Model transformation part 1 (Sec. V): We systematically derive a *role-based architecture* following the HAM pattern for the initially specified self-adaptive component.

(4) Specification of application logic: After the first transformation, we rely on the role-based architecture to specify the application logic of behavioral modes as *mode role behaviors*.

(5) Model transformation part 2 (Sec. VI): We integrate the mode role behaviors and the adaptation automaton into the

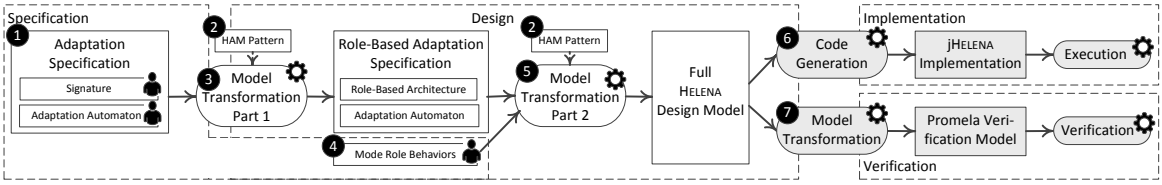


Fig. 1. The HELENA development process for self-adaptive systems. Rectangular boxes denote artifacts, boxes with rounded edges are activities. The boxes marked with a user icon are artifacts which have to be created by the specifier; all other artifacts can be systematically derived by the shown activities.

role-based architecture to gain a *full* HELENA *design model*.

(6) Implementation: The execution framework jHELENA [16] transfers the concept of roles to Java. We rely on this framework to implement and execute our HELENA model. An automatic code generator from HELENA specifications to jHELENA code was already presented in [17].

(7) Verification: Additionally, HELENA models can be formally analyzed. In [18], we introduced model-checking to HELENA by systematically transforming HELENA models to Promela to be able to verify temporal properties using Spin.

II. Search-And-Rescue Scenario

One of the case studies of the EU-project ASCENS (`www.ascens-ist.eu`) is a robotic search-and-rescue scenario [19]. Robots are distributed over an unknown area where recently a disaster happened. The robots have to find victims and to transport them to a rescue area. Since humans should not enter the dangerous area, the robots have to self-adaptively manage their behaviors. For example, during searching for victims, a robot searches randomly and informs other robots about the location of found victims. If it was informed about the victim's location by another robot, it switches to a directed walk towards the victim. A robot also changes its behavior whenever it reaches a victim during search and starts to help rescuing the victim. Orthogonally, the robot may run out of battery at any time. Then, the robot switches to a low-power behavior and waits for another robot to get recharged.

III. Adaptation Specification

The first step in the development process in Fig. 1 is the adaptation specification. A self-adaptive system is composed of a set of self-adaptive components given by their *signature* and *adaptation automaton*. The automaton describes the rules how a component changes its current behavioral mode depending on perceptions. Formally, an *adaptation specification* is a pair $AdapSpec = (sigs, auts)$ such that $sigs$ is a set of signatures of self-adaptive component types and $auts$ is a set of adaptation automata. For every self-adaptive component type $ct \in sigs$, exactly one automaton $AA_{ct} \in auts$ must exist.

A. Signature of Self-Adaptive Component Types

The signature $ct = (nm, attrs, attrs_{aware}, modes)$ of a self-adaptive component type describes the static properties of the component. It has a name nm and stores normal data (attributes of the set $attrs$). Additionally, it stores perceptions about the environment and its own state as awareness data (attributes of the set $attrs_{aware}$). While component data may also be perceptions, it does not directly trigger adaptation. However, it may transitively influence awareness data which is then the ultimate source to decide whether to adapt. The set $modes$ is a set of behavioral modes which the component can switch in response to changes of awareness data.

Example: In our search-and-rescue scenario, we just need one self-adaptive component type. Its signature is shown in Fig. 2 in graphical notation. The type `Robot` stores its own position, is aware of its battery level and whether it is at a victim's position (the position is not awareness data; although it is perceived, it does only transitively trigger self-adaptation when reaching a victim). It is aware of the environment by storing the position of a victim (`null` if unknown), and whether it was requested as recharger by another robot. A robot has five behavioral modes as in explained in Sec. II: `RandomWalk`, `DirectedWalk`, `Rescue`, `LowBattery`, and `Recharge`.

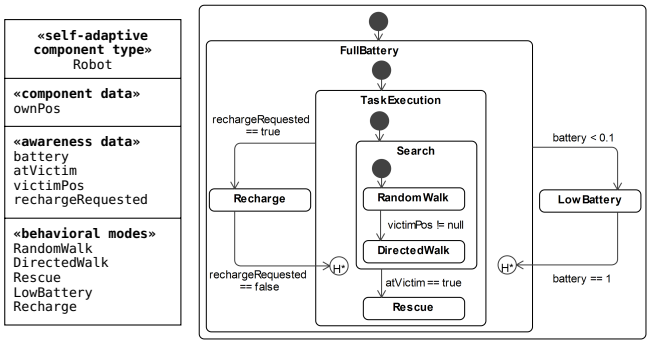


Fig. 2. Signature and adaptation automaton of a robot

B. Adaptation Automata

To describe the rules for switching behavioral modes, we propose *hierarchical labeled transition systems* which extend standard labeled transition systems by hierarchy and history.

Auxiliary definitions: A set Q of states over Q_{basic} consists of all basic states $q \in Q_{basic}$ and all complex states $q = (cset, init)$ such that $cset \subseteq Q$ is a finite, non-empty set of states and $init \in cset$ is the initial state of the set. We denote the set of all basic states transitively included in a state q by *basic-states*(q) and the set of all sub-states (of any depth) by *sub-states*$^*(q)$. A state q is *well-formed* if it is a basic state $q \in Q_{basic}$ or it is a complex state $q = (cset, init)$ such that all $q' \in cset$ are well-formed and all sets *basic-states*(q') are disjoint.

H-LTS: A *hierarchical labeled transition system (H-LTS) over a set Q_{basic} of basic states* is a tuple (q, L, δ, δ^*) with

- a state q from the set of (well-formed) states over Q_{basic},
- a set L of labels,
- transition relations $\delta, \delta^* \subseteq$ *sub-states*$^*(q) \times L \times$ *sub-states*$^*(q)$.

The H-LTS is composed of only one state from the states over Q_{basic}, the *core state*. Labeled transitions connect two sub-states (of any depth) of this state. Semantically, an H-LTS is represented by a standard LTS. Transitions in δ originating from a complex state are an abbreviation for adding a transition with the same target to every basic sub-state of the origin. Transitions in δ leading to a complex state actually target the initial state of the complex state. A separate set δ^* of history transitions reflects the idea of deep history states in UML state charts. A history transition leading to a complex state q means returning to the last visited basic state $q' \in \textit{basic-states}(q)$.

Adaptation automata: An *adaptation automaton* for a self-adaptive component type is a special instance of an H-LTS. The basic states range over all different behavioral modes of the component type. Transitions between states are initiated by predicates over awareness data of the component type.

Example: Fig. 2 shows the adaptation automaton of a robot in a graphical notation similar to UML state charts. We just highlight complex states and history transitions. The two search strategies `RandomWalk` and `DirectedWalk` are integrated into one complex state `Search`. Thus, we can express that independently from the search strategy, the robot switches to rescuing as soon as it is at a victim's position (transition from `Search` to `Rescue`). Similarly, the robot interrupts its current behavior if it goes out of battery (transition from `FullBattery` to `LowBattery`). However, it resumes the previously executed behavior upon recharge expressed by the history transition from `LowBattery` back to the history state of `FullBattery`.

C. Benefits

Conceptually, the adaptation specification realizes the idea of separation of concerns. The transitions of an adaptation automaton are triggered by changes of awareness data and thus capture adaptation logic only. The states of the automaton are the behavioral modes of a component and serve as placeholders where concrete application logic can be plugged in.

Methodologically, hierarchical states allow subsuming transitions with the same trigger from all sub-states of a complex state into one single transition. History transitions represent returning to the last visited sub-state of a complex state and therefore prevent unfolding the adaptation automaton for every possible last visited sub-state. Thus, adaptation rules are specified more compactly than in a standard LTS.

IV. Role-Based Adaptation Design Pattern

In the development process in Fig. 1, we rely on the HAM pattern to derive a HELENA design model from an adaptation specification. It is a design pattern which proposes to realize SAS with a role-based architecture based on HELENA [15] according to the autonomic manager pattern [9].

A. Autonomic Manager Pattern

In the autonomic manager pattern (cf. Fig. 3a), an adaptable component is managed by an adaptation manager. The manager monitors the environment and the component itself, analyzes and plans appropriate reactions, and executes adaptations on the managed component (cf. MAPE-K loop [4]). Thereby, the component takes perceptions about the environment with its sensor. It forwards these perceptions together with observations about its own state via its emitter to the manager. Therefore, the adaptation manager observes the state of the component and transitively of the environment via its sensor. Depending on these perceptions, it internally decides about appropriate reactions and imposes them via its effector on the component. The component realizes the instructed adaptations affecting the environment through its effector.

However, the pattern lacks a concept how the component actually changes its behavior. We think that roles which can be played by components are an intuitive representation of context-specific behavior and can therefore provide the necessary concept for switching between behavioral modes.

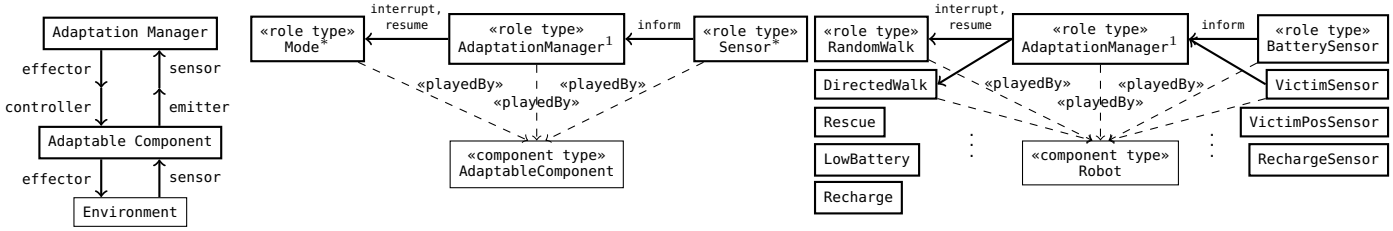
B. HELENA *in a Nutshell*

The role-based modeling approach HELENA [15] provides concepts to describe systems where components play certain roles in ensembles to perform global goal-oriented tasks. A role encapsulates context-specific capabilities and behavior. By switching between roles, the component changes its currently executed behavior. By adopting several roles in parallel, it concurrently executes different behaviors.

Ensemble structures: The foundation for ensemble-based systems are components characterized by their type. They have a name and a set of attributes representing basic information that is useful in all roles the components can adopt. They also provide a set of operations which can be invoked by their roles. Roles are classified by role types. Given a set CT of component types, a *role type* rt is a tuple $(nm, compTypes, roleattrs, rolemsgs)$: nm is the name of the role type; the set $compTypes \subseteq CT$ determines the component types which can adopt the role; the set $roleattrs$ allows to store data that is only relevant for performing the role; the set $rolemsgs$ determines the incoming and outgoing messages supported by the role. To define the structural characteristics of collaborations, an *ensemble structure* specifies the role types whose instances form the ensemble and determines how many instances of each role type may contribute to the ensemble by a multiplicity (like $0..1$, 1, $*$, $1..*$ etc.) for each type. We assume that between two role types the messages which are output on one side and input on the other side can be exchanged.

Ensemble specifications: The *behavior* of a role is given by a process expression built from the null process, action prefix, guarded choice (branch is nondeterministically selected if several branches are executable), and process invocation [18]. Guards are predicates over component or role attributes. There are actions for creating and retrieving role instances, sending or receiving messages, and invoking an operation of the owning component. These actions must fit to the declared ensemble structure, e.g., messages can be only sent by roles which declare them. A collaboration is specified by an *ensemble specification* consisting of an ensemble structure Σ and a set of role behaviors, one for each role type occurring in Σ.

Semantics: Ensemble specifications are semantically interpreted by labeled transition systems, i.e., ensemble automata



(a) Aut. Manager pattern [9] (b) HELENA Adaptation Manager (HAM) pattern (c) Applying the HAM pattern to the search-and-rescue scenario

Fig. 3. Adaptation design patterns

[15], [18]. Ensemble states capture the currently existing role instances with their data and control states. Transitions between ensemble states are triggered by role instance creation or retrieval, communication actions, and operation calls.

C. HELENA *Adaptation Manager Pattern*

We propose the HAM pattern (cf. Fig. 3b) to realize SAS by a role-based design. It reuses the ideas from the autonomic manager pattern to separate monitoring and adaptation logic from the adaptable component, but augments it by roles which the component switches to realize the instructed adaptations.

Adaptable component: The central entity of the HAM pattern is the adaptable component. Since we apply the idea of roles, the component itself is not active, but is just a data container and serves as the execution platform for roles which are the active entities providing context-specific behavior.

Behavioral modes: A component adapts its behavior by switching between roles representing behavioral modes. The abstract role type `Mode` in Fig. 3b is instantiated by concrete mode role types for every behavioral mode. Although there can be many different mode role types, the component should only adopt *one mode role* at a time[1]. The adaptation manager has to take care of deactivating the mode roles which are not adequate anymore with the message *interrupt* and activating an appropriate mode role with the message *resume*.

Sensors: To decide which mode role should be activated, the component has to monitor its awareness data. We externalize monitoring to sensors represented by roles again. The abstract role type `Sensor` in Fig. 3b is instantiated by concrete sensor types for every item of awareness data[1]. There are many different sensor roles similarly to mode roles, but the component needs to adopt *all sensor roles* in parallel to monitor all items of awareness data. Each sensor is responsible to continuously inform the adaptation manager about the value of its monitored awareness data item by sending an *inform* message.

Adaptation manager: The `AdaptationManager` is yet another role type on top of the adaptable component which can only be instantiated once per component. It is responsible for realizing the adaptation logic. That means, it continuously receives the sensor data via `inform` messages, internally decides how to react to these perceptions, and switches the currently active mode role by sending `interrupt` and `resume` messages.

[1]Note that it is not part of the pattern how the behaviors of all introduced roles can be derived. However, in Sec. VI we explain how the specifier adds behaviors for mode roles and how all other role behaviors can systematically be derived from an adaptation specification (cf. Sec. III).

Example: Let us illustrate at our search-and-rescue example how we apply the HAM pattern to derive a role-based HELENA model (cf. Fig. 3c) for the self-adaptive component type `Robot` in Fig. 2. (i) The central entity in the design model is the robot itself. It is a component type, i.e., it is just the resource for executing the different behavioral modes. (ii) Each behavioral mode in the specification is represented by a new role. For example, the robot is able to execute a behavioral mode like randomly searching for a victim by adopting the corresponding role `RandomWalk`. (iii) For every item of awareness data in the specification, the corresponding sensor is represented by a new role, e.g., the sensor role `BatterySensor` for the awareness attribute `battery`. (iv) Finally, the adaptation manager is installed as a role on top of the robot[1].

D. *Benefits*

(1) The proposed architecture respects separation of concerns as advocated in the autonomic manager pattern. (2) Roles intuitively express the different tasks of self-adaptive components, like being aware of the environment, managing adaptation, and performing particular behavioral modes. (3) We can automatically derive a formal HELENA model from an adaptation specification following the HAM pattern and equip it with application logic for each behavioral mode role (cf. Sec. V and Sec. VI). (4) Relying on the formal foundation of HELENA allows to analyze the derived model for communication errors and goal satisfaction [18]. With HELENATEXT [17] and jHELENA [16], it is also possible to automatically generate a Java implementation for the HELENA model and execute it.

V. MODEL TRANSFORMATION PART 1

Input Artifacts: The first model transformation in the HELENA development process starts from the adaptation specification of self-adaptive component types consisting of the signature and the adaptation automaton.

Output Artifacts: From the adaptation specification, a role-based architecture in HELENA is systematically derived which follows the HAM pattern. It consists of the component with attributes and operations as well as roles for behavioral modes, sensors, and the adaptation manager. We only exemplify the translation at our search-and-rescue scenario. The full translation is described in a technical report [20]. However, to gain a full HELENA model, the role-based architecture has to be extended by behaviors for all roles which is pursued in the second model transformation (cf. Sec. VI).

Example: This model transformation translates the adaptation specification in Fig. 2 to the role-based architecture in Fig. 3c. (i) The robot is reflected by the component type ($Robot$, { *ownPos*, *batteryLevel*, *atVictim*, *victimPos*, *rechargeReq* }, {*updateBL*, *updateAV*, *updateVP*, *updateRR*}). We no longer distinguish (normal) data and awareness data attributes, but for each item of awareness data we add a special update-operation which updates the value according to the current perceptions. (ii) For each behavioral mode, we create a mode role which can only receive the messages *interrupt* and *resume*, e.g., the mode role ($RandomWalk, \{Robot\}, \emptyset, \{interrupt, resume\}$). (iii) For each awareness data attribute, we create a sensor role which can only send the message *inform*, e.g., the sensor role ($BatterySensor, \{Robot\}, \emptyset, \{inform\}$). (iv) The adaptation manager ($AdaptationManager, \{Robot\}, \emptyset, \{interrupt, resume, inform\}$) is yet another role which takes care to adapt the component.

VI. Model Transformation Part 2

Input Artifacts: The input for the second model transformation is the role-based architecture which was created during the first model transformation, the adaptation automaton for the self-adaptive component type from the initial adaptation specification, and mode behaviors which define the application logic in each behavioral in the form of HELENA role behaviors.

Output Artifacts: The transformation completes the role-based architecture with a behavior for each role gaining a fully specified HELENA model. The role behaviors are derived such that the component exhibits the desired adaptive behavior (the formal translation is described in [20]): The adaptation manager is informed by the sensors about perceptions. It internally decides about appropriate adaptations based on the adaptation automaton and deactivates and activates the corresponding mode roles. The activated mode role takes care to execute its associated application logic.

Example: For our search and rescue example, this transformation takes as input the role-based architecture in Fig. 3c, the adaptation automaton in Fig. 2, and role behaviors for each mode role (not shown here). (i) For the robot, no behavior is generated since it just adopts its associated roles. (ii) The behavior of each mode role is extended such that the manager can control its execution. Firstly, the behavior is initially paused and is just started upon request with the message *resume* from the manager. Secondly, the role behavior always needs to be interruptible and resumable to allow to switch the currently executed mode. Thus, for example, the role behavior $RB_{RandomWalk} =$ **owner**.*randomStep* . $RB_{RandomWalk}$ is modified to

$$
\begin{aligned}
RB_{RandomWalk} &= ?resume.RW \\
RW &= \textbf{if } (\textbf{true}) : \{?interrupt.?resume\ .\ RW\} \\
&\quad\ \textbf{or } (\textbf{true}) : \{\textbf{owner}.randomStep\ .\ RW\}
\end{aligned}
$$

(iii) Intuitively, the role behavior of a sensor takes care to continuously advise the owning component to update the value of the monitored awareness data attribute and to send the new value to the adaptation manager. Thus, for example, the role behavior of the sensor role *BatterySensor* is given by

$$
\begin{aligned}
RB_{BatterySensor} &= am \leftarrow \textbf{create}(AdaptationManager, \textbf{owner})\ .\ BS \\
BS &= \textbf{owner}.updateBL\ .\ am!inform(\text{“}battery\text{”}, \textbf{owner}.battery)\ .\ BS
\end{aligned}
$$

(iv) The adaptation manager is equipped with a behavior that is responsible for changing the currently active role of the managed component according to the adaptation automaton. We translate the adaptation automaton in three steps into a role behavior: flattening the automaton to an LTS, deriving a process term from the LTS, adding initialization of mode roles. Flattening is rather involved since hierarchy and history needs to be resolved [21]. Deriving a process term follows the idea of deriving a right linear grammar from a nondeterministic finite automaton [22]. Let us exemplify the result of complete derivation by an excerpt of the behavior of the manager (we use the notation **else** in the usual meaning as an abbreviation).

$$
\begin{aligned}
RB_{AdapationManager} = \quad & rand \leftarrow \textbf{create}(RandomWalk, \textbf{owner})\ . \\
& \ldots\ low \leftarrow \textbf{create}(LowBattery, \textbf{owner})\ . \\
& rand!resume\ .\ Rand \\
Rand = \quad & ?inform(attr,val)\ . \\
& \textbf{if } (attr{=}{=}\text{“}battery\text{”}\ \&\ val{<}0.1)\ : \\
& \quad \{rand!interrupt\ .\ low!resume\ .\ LowFromRand\} \ldots \\
& \textbf{or } (\textbf{else})\ : \{Rand\} \\
LowFromRand = \quad & ?inform(attr, val)\ . \\
& \textbf{if } (attr{=}{=}\text{“}battery\text{”}\ \&\ val{=}{=}1)\ : \\
& \quad \{low!interrupt\ .\ rand!resume\ .\ Rand\} \ldots
\end{aligned}
$$

First, instances for all behavioral mode roles are created and the behavior of *RandomWalk* is started since it is initial in the adaptation automaton in Fig. 2. According to the adaptation automaton, the manager then has to wait that it got informed about a victim's position, it found a victim, it was requested as a recharger by another robot, or it runs out of battery where we only show the latter in the role behavior. Whenever one of these conditions becomes true, the manager changes the currently active role. For example, when the battery is low, it interrupts the role *RandomWalk* and resumes the role *LowBattery*. Afterwards, the manager continues its role behavior in a state where it knows that the current role is *LowBattery* and the previous role was *RandomWalk* represented by *LowFromRand*.

VII. Related Work

According to FORMS [23] the distinguishing characteristics of SAS is the capability of reflection about itself. Our methodology is aligned with the idea of reflection since the adaptation manager decides on the adequacy of the currently executed behavioral mode separately from the application logic. Furthermore, we consider our methodology as an architecture-based self-adaptation approach like [12]–[14] since the adaptation manager reflects on the architecture of the self-adaptive component in the sense of currently executed behavioral mode. In contrast, we propose a role-based architecture for switching between behavioral modes to adapt a self-adaptive component.

Adaptation Specification Techniques: Luckey and Engels [5] specify adaptation logic in adapt-cases (similarly to use-cases) on top of a system's architecture. The operationalization is based on UML activity diagrams and can be checked against quality properties. Opposed to them, we specify adaptation logic similarly to UML state charts since they provide hierarchically composed states and history states. For implementation, the authors do not introduce any first-class concepts to realize adapt-cases while we propose a

specific role-based design to transfer the separation of adaptation logic and application logic to implementation. Automata-based approaches [6]–[8], [24] specify adaptation by evolution of finite state machines representing behavioral modes. We augment these approaches by H-LTS with history and hierarchy which allow to specify adaptation rules more compactly. Additionally, we can systematically derive verification and implementation models due to relying on HELENA.

Role-Based Adaptation: Steegmans et al. [10] propose a design process for adaptive agents based on roles. They realize the role model by free-flow trees and a corresponding framework. In contrast, we transfer the concept of roles to the implementation to preserve a clean architecture. Self-Epsilon [11] proposes to employ a controller on each object in a self-adaptive system. The controller takes care to change the role of the object depending on the current context. While conceptually similar to our approach, Self-Epsilon does not aim at providing a formal role-based model for reasoning or at explicitly describing the architecture of such systems.

VIII. CONCLUSION

We presented a holistic development process how to engineer SAS separating adaptation logic from application logic. Key concept is to realize behavioral modes by roles which a component can dynamically adopt. We propose adaptation automata as a rich specification technique of adaptation logic and the HELENA Adaptation Manager pattern to realize a self-adaptive system by a role-based architecture in HELENA.

Discussion: Though powerful and compact, an H-LTS is more complex than a standard LTS. Thus, we intend to provide a graphical representation similarly to UML state charts. Furthermore, roles are an intuitive concept for encapsulating context-specific behavior, but roles cannot share behavior, e.g. obstacle avoidance in our robotic example. Thus, we plan to introduce hierarchy of roles similar to the subsumption architecture [25] and concurrent execution of mode roles.

Future work: Similarly to the high-level change management in ActivFORMS [26] or PSCEL [27], the adaptation strategy could be controlled by hierarchically composing adaptation managers or techniques from artificial intelligence could generate adaptation logic from a domain specification or observations. We also consider collective adaptation [28] and adaptive collaborations [29] as a very interesting extension.

ACKNOWLEDGMENT

We thank Benedikt Hauptmann, Rolf Hennicker, Philip Mayer, Andreas Vogelsang, and Danny Weyns for very valuable feedback on the proposed methodology. Furthermore, we are grateful for detailed comments of anonymous reviewers.

REFERENCES

[1] B. H. C. Cheng *et al.*, "Software Engineering for Self-Adaptive Systems: A Research Roadmap," in *Software Engineering for Self-Adaptive Systems*, ser. LNCS, vol. 5525. Springer, 2009, pp. 1–26.

[2] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *Trans. Auton. Adap. Systems*, vol. 4, no. 2, 2009.

[3] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, "Engineering Self-Adaptive Systems Through Feedback Loops," in *Int. Sym. Software Engineering for Self-Adaptive Systems*. Springer, 2009, pp. 48–70.

[4] IBM Corporation, "An architectural blueprint for autonomic computing," 2006, http://goo.gl/5Lo5AO.

[5] M. Luckey and G. Engels, "High-Quality Specification of Self-Adaptive Software Systems," in *Int. Sym. Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2013, pp. 143–152.

[6] Y. Zhao, D. Ma, J. Li, and Z. Li, "Model Checking of Adaptive Programs with Mode-extended Linear Temporal Logic," in *Int. Conf. Engineering of Autonomic and Autonomous Systems*. IEEE, 2011, pp. 40–48.

[7] E. Merelli, N. Paoletti, and L. Tesei, "A Multi-Level Model for Self-Adaptive Systems," in *Int. Workshop Foundations of Coordination Languages and Self Adaptation*, vol. 91. EPCTS, 2012, pp. 112–126.

[8] R. Bruni, A. Corradini, F. Gadducci, A. Lluch Lafuente, and A. Vandin, "Adaptable Transition Systems," in *Recent Trends in Algebraic Development Techniques*, ser. LNCS. Springer, 2013, vol. 7841, pp. 95–110.

[9] M. Puviani, G. Cabri, and F. Zambonelli, "A Taxonomy of Architectural Patterns for Self-adaptive Systems," in *Int. C* Conf. Computer Science and Software Engineering*. ACM, 2013, pp. 77–85.

[10] E. Steegmans, D. Weyns, T. Holvoet, and Y. Berbers, "A Design Process for Adaptive Behavior of Situated Agents," in *Int. Conf. Agent-Oriented Software Engineering*. Springer, 2005, pp. 109–125.

[11] S. Monpratarnchai and T. Tetsuo, "Applying Adaptive Role-Based Model to Self-Adaptive System Constructing Problems: A Case Study," in *Int. Conf. Eng. Autonomic and Autonomous Systems*, 2011, pp. 69–78.

[12] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An Architecture-Based Approach to Self-Adaptive Software," *Intelligent Systems*, vol. 14, no. 3, pp. 54–62, 1999.

[13] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[14] J. Kramer and J. Magee, "Self-Managed Systems: An Architectural Challenge," in *Future of Softw. Eng.* IEEE, 2007, pp. 259–268.

[15] R. Hennicker and A. Klarl, "Foundations for Ensemble Modeling - The Helena Approach," in *Specification, Algebra, and Software*, ser. LNCS, vol. 8373. Springer, 2014, pp. 359–381.

[16] A. Klarl and R. Hennicker, "Design and Implementation of Dynamically Evolving Ensembles with the Helena Framework," in *Australasian Software Engineering Conf.* IEEE, 2014, pp. 15–24.

[17] A. Klarl, L. Cichella, and R. Hennicker, "From Helena Ensemble Specifications to Executable Code," in *Int. Sym. Formal Aspects of Comp. Software*, ser. LNCS, vol. 8997. Springer, 2015, pp. 183–190.

[18] A. Klarl, R. Hennicker, and M. Wirsing, "Model-Checking Helena Specifications with Spin," in *Festschrift for Jose Meseguer*, ser. LNCS. Springer, submitted 2015. [Online]. Available: http://goo.gl/gyysJm

[19] R. D. Nicola, M. Loreti, R. Pugliese, and F. Tiezzi, "A Formal Approach to Autonomic Systems Programming: The SCEL Language," *Trans. Auton. Adap. Systems*, vol. 9, no. 2, pp. 7:1–7:29, 2014.

[20] A. Klarl, "Engineering Self-Adaptive System with Role-Based Architectures in Helena," Ludwig-Maximilians-Universität München, Germany, Tech. Rep., 2015. [Online]. Available: http://goo.gl/gyysJm

[21] X. Devroey, G. Perrouin, M. Cordy, A. Legay, P. Schobbens, and P. Heymans, "State Machine Flattening: Mapping Study and Assessment," *Computing Research Repository*, vol. abs/1403.5398, 2014.

[22] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[23] D. Weyns, S. Malek, and J. Andersson, "FORMS: Unifying Reference Model for Formal Specification of Distributed Self-Adaptive Systems," *Trans. Auton. Adap. Systems*, vol. 7, no. 1, pp. 8:1–8:61, 2012.

[24] J. Zhang and B. H. C. Cheng, "Model-based development of dynamically adaptive software," in *Int. Conf. Softw. Eng.* ACM, 2006, pp. 371–380.

[25] R. A. Brooks, "A Robust Layered Control System For a Mobile Robot," *Robotics and Automation*, 1986.

[26] M. U. Iftikhar and D. Weyns, "ActivFORMS: Active Formal Models for Self-Adaptation," in *Int. Sym. Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2014, pp. 125–134.

[27] A. Margheri, R. Pugliese, and F. Tiezzi, "Linguistic Abstractions for Programming and Policing Autonomic Computing Systems," in *Int. Conf. Autonomic and Trusted Computing*. IEEE, 2013, pp. 404–409.

[28] M. Puviani, G. Cabri, and L. Leonardi, "Enabling Self-expression: the Use of Roles to Dynamically Change Adaptation Patterns," in *Int. Conf. Self-Adaptive and Self-Organizing Systems*. IEEE, 2014, pp. 14–19.

[29] H. Zhu, M. Zhou, and M. Hou, "Adaptive Collaboration Based on the E-CARGO Model," *Agent Technol. Syst.*, vol. 4, no. 1, pp. 59–76, 2012.