Behavioral Modeling of IT Ecosystems

Ursula Goltz, Narges Khakpour, Christoph Knieke, Lukas Märtin

August 9, 2010

This work was funded by the NTH School for IT Ecosystems. NTH (Niedersächsische Technische Hochschule) is a joint university consisting of Technische Universität Braunschweig, Technische Universität Clausthal, and Leibniz Universität Hannover.
Abstract

IT ecosystems - systems composed of a large number of distributed, autonomous, cooperating, decentralized, interacting, organically grown, heterogeneous, and continually evolving subsystems - are the future system generation. Today’s state of the art does not enable us to develop these systems. Within the NTH Focused Research School for IT Ecosystems, research project AIM deals with methods and tools to guarantee the functionality of a complex IT ecosystem especially when a top-down design is not possible anymore. Thus, adaptive information- and collaboration architectures considering independent evolution of subsystems as well as suitable control mechanisms are examined.

This technical report analyzes how adaptive behavior of subsystems can be modeled adequately by standard formalisms for behavioral modeling (e.g. UML) as well as advanced approaches for modeling adaptive behavior (e.g. PobSAM). We apply the selected modeling languages on a fictional case study, an airport departure scenario. The smart airport itself can be seen as an IT ecosystem due to the complexity of the interacting systems.
Contents

List of Figures vi

1. Introduction 1

2. Basics 2
   2.1. Autonomic Computing 2
   2.2. Meta-Model of the Agent Architecture 4

3. Modeling Adaptive Behavior 6
   3.1. Live Sequence Charts 6
   3.2. Statecharts 9
   3.3. Activity Diagrams 10
   3.4. PobSAM 17

4. Case Study 19

5. Conclusion 24

A. Modeling Tools 26
   A.1. Play-Engine 26
   A.2. Rhapsody 27
   A.3. LAD-Simulation 28

References 29
# List of Figures

1. Closed control cycle of an autonomic element .......................... 3
2. MAPE cycle ................................................................. 4
3. Meta-model for agent architecture ........................................ 5
4. LSC specifying a traffic light phase ...................................... 7
5. Prototypical GUI of a set of traffic lights at the crossing “Hamburger Straße/Rebenring” ..................................................... 8
6. Statechart of a MAPE cycle ............................................... 9
7. Statechart of the monitoring phase ....................................... 10
8. Graphical elements of activity diagrams ................................. 11
9. LAD model for Lego Mindstorms NXT ................................. 13
10. Case study: “Line Follower” .............................................. 15
11. Use case diagram “Airport Departure” .................................. 20
12. Grid of the smart airport .................................................. 22
13. Activity diagram “Transport Request” ................................... 23
14. Activity diagram “Execute Request” .................................... 23
15. Connection to Repast Simphony .......................................... 24
16. PLAY-ENGINE tool .......................................................... 26
17. RHAPSODY tool ............................................................ 27
18. LAD-Simulation tool ........................................................ 28
1. Introduction

Next generation software-intensive IT systems can no longer be designed in a purely top-down fashion. They are complex systems of systems, where the individual systems and components are modeled, built, operated, and controlled by different stakeholders, across organizations. Also, software systems and components are equipped with increasing autonomy, including capabilities for self-configuration and self-organization. The main research question addressed in the IT ecosystems project is how we can support this autonomy with appropriate software architectures, methods, and tools. At the same time the overall functioning of future software intensive systems, supporting their controllability and dependability, and, for critical applications, enforcing required guarantees regarding non-functional and functional aspects of system behavior has to be maintained.

Within the NTH Focused Research School for IT Ecosystems, research project AIM contributes to the IT ecosystems vision by studying models, architecture, methods, tools, and applications of autonomous agents in organized localities. AIM investigates mechanisms for information exchange, coordination, and cooperation between humans and computerized artifacts (software applications and tools, but also e.g. autonomous vehicles or virtual characters). This requires us to understand the processes of local adaptation, modification, and evolution of individual systems, and the requirements for control and regulation to enable proper interworking of the individual systems in the IT ecosystem.

Our work package of the AIM project focuses on model based analysis and design of system of systems in the context of IT ecosystems. Each subsystem of an IT ecosystem has to provide different levels of flexibility that can be classified as adaptability, modifiability, and evolution. We aim at developing a modeling approach supporting the adaption, modification, and evolution of subsystems within an IT ecosystem, concentrating on functional behavior aspects. The challenge is to combine concepts for adaptability and evolvability from abstract approaches with more standard modeling techniques, extending them where necessary, thus providing a meta-model for functional behavior of individuals in IT ecosystems covering adaptiveness and evolution. We will establish a sound semantic basis and prototypical tool support. Based on the resulting modeling approach we will investigate online (monitoring) and offline analysis techniques.

In this technical report, we analyze existing modeling approaches for functional behavior with respect to their suitability for IT ecosystems. Based on an airport departure case study, we investigate how adaption can by supported. Certain behavior models of UML (e.g. activity diagrams) [29] however need to be semantically clarified and to be evaluated concerning their expressiveness. We also analyze PobSAM (Policy-based Self-Adaptive Model) [21] as a flexible model, for developing, specifying, and verifying self-adaptive systems.

The paper is organized as follows. The next section gives an overview on adaptive systems and the meta-model of the agent architecture. In Section 3 we discuss three
standard modeling languages with respect to their capabilities for modeling adaption in the context of IT ecosystems. The results of the case study are presented in Section 4. Section 5 concludes.

2. Basics

In the context of adaptability the subject areas Self Management and the AIM meta-model of the agent architecture will be described in this section.

2.1. Autonomic Computing

The scale, distribution, and interaction of autonomous systems steps up continuously. This leads to an increase in complexity and networking. In order to handle these new challenges adequate mechanisms for automated management systems are implemented. The maintenance of systems is significantly reduced by systems that can be managed almost completely by themselves [9]. The management of systems is performed by self management.

Self Management

Self management complies with fundamental processes in the human organism that are not consciously controlled by the human, e.g. heartbeat and respiration. The IBM Autonomic Computing Initiative divides self management into four subitems [20, 16]:

- Self-configuration is used to adapt to changing environments through automatic reconfiguration
- Self-healing allows the detection, diagnostics and disposal of faults
- Self-optimization monitors and varies resources according to the operational and customer requirements
- Self-protection forecasts, identifies and prevents attacks on the system

Autonomic systems can adapt to new environments and respond to changes through self management. In particular, the coordination and adaption between different systems takes center stage. Hence, the systems have to operate adaptively.

In the area of organic/autonomic computing, several architectures for supporting self-organizing/adaptive behavior have been proposed, e.g.

- The MAPE cycle (Monitor, Analyzer, Planner, and Executive) [20]: The basic elements of an adaptive system are described as part of a specialized control cycle.
Observer/Controller Architectures from organic computing [31]: The system under observation and control (SuOC) consists of a set of interacting intelligent autonomous units. The observer measures, analyzes, and reports the system behavior to the controller. The controller applies adequate actions to the SuOC to achieve a given goal.

Operator/Controller Modules developed by the DFG SFB 614 [28]

Agent-based approaches (e.g. [32])

Autonomous System-On-Chip Architectures (e.g. [2])

Furthermore, in a variety of research projects the priority research program SPP 1183 of the German Research Foundation (DFG) addresses fundamental challenges in the design of organic computing systems; its objective is a deeper understanding of emergent global behavior in self-organizing systems and the design of specific concepts and tools to support the construction of organic computing systems for technical applications.

Designing self-optimizing systems based on UML is addressed by the Mechatronic UML development approach [35, 3] which combines domain specific modeling and refinement techniques with verification. By means of Mechatronic UML reconfiguration of hierarchical component systems can be described in a modular way. The use of Mechatronic UML is supported by a tool called FUJABA (From UML to Java and Back Again, see http://www.fujaba.de).

As an example of an approach from autonomic computing, we will describe the MAPE cycle in more detail in the following. In section 3.2, we give an example of a self manager including a MAPE cycle, implemented by statecharts.

**MAPE cycle**

Each autonomic element consists of a managed element. A managed element represents a single resource or a combination of multiple resources. An autonomic manager administers (manages) these elements. The control of an autonomic element is realized by a closed control cycle (see Figure 1, adapted from [9]). Closed control cycles are hypothetically able to control a system so that the system always remains in a desired state without interference from outside.

![Figure 1: Closed control cycle of an autonomic element](http://www.fujaba.de)
An autonomic manager is responsible for the retrieval of autonomic behavior of a managed element. It is predicated on four integrated phases and a knowledge base.

- Monitor collects and evaluates the measurement values from the managed element,
- Analyzer interprets these values to identify the current situation of the managed element. In case of failure it recommends possible recovery actions,
- Planner schedules these actions to fit the business requirements, and
- Executive initiates the suggested actions in the managed element.

As shown in Figure 2 these four phases are the basis of the MAPE cycle [10, 20].

![Figure 2: MAPE cycle](image)

The phases share one knowledge base to evaluate measurements and make decisions according to given rules. Knowledge could be e.g. policies, or logs in a repository.

### 2.2. Meta-Model of the Agent Architecture

As a basis for behavioral description, either through modeling or implementation, the AIM meta-model and architecture group [17] developed a meta-model providing a template for the architecture of a single software agent. Figure 3 shows the four-layer model.

According to the specifications of a layer model, the layers build from the bottom to the top of each other. The tasks of the single layers can be described roughly as follows:

- Mechatronic layer
  - Mechanical and electronic properties of the agent
- Robust execution layer
  - Execution of single control functions (feedback control problems)
  - Implementation of reactive behavior to respond directly on environmental changes
  - Consideration of various alternatives
- Subjective context layer
– Individual consideration of the environment by a single agent
– Adaptation of behavior according to individual world view, goals and plans
– Based on individual knowledge (data) and program logic
– Teaching ability of the agent allows adaptation
– Data are obtained by examining and surveillance
– Environment is influenced by feedback (actions)

• Social context layer
  – Integrated view of the environment (including other agents) by a single agent respecting common norms, goals and plans
  – Based on the knowledge of several agents and their logic
  – Coordination below the agent allows adaptation

The upper two layers are divided into three parts according to a BDI architecture (Belief-Desire-Intention [30]). Belief is perceived as the state of the environment. At the top level this state is defined by Norms. In the subjective context the state is derived from the behavior of other agents and the environment. The goals of all agents (social
context) or of single agents (subjective context) is classified as Desire. Intention includes concrete plans for implementation of individual or common goals, while respecting the environmental conditions.

More information on the structure of the agent architecture can be found in [17]. In particular, in the subjective context adaptive modeling plays a leading role.

3. Modeling Adaptive Behavior

In this section, different approaches for modeling adaptive behavior are discussed using concrete case studies, including a comparison between the approaches. We have applied Live Sequence Charts (LSC) and Statecharts on a traffic light case study. The behavior of an existing traffic light system at a crossing in Braunschweig (Germany) is specified by an LSC/Statechart model. In addition, we extend the model so that the traffic light system adapts to the current traffic situation by adjusting the duration of the traffic light phases, and thus optimizing the traffic flow.

To demonstrate adaption in the context of mobile robots, we have specified the behavior of Lego Mindstorms NXT robots by a set of activity diagrams. By means of an integrated Eclipse plug-in the NXT robots are controlled according to the behavior specified by the activity diagram model.

3.1. Live Sequence Charts

The inter-object, scenario-based language of LSCs thoroughly defined in [13] is a visual formalism extending MSCs. LSCs offer the possibility to distinguish between possible and mandatory behavior which is modeled by existential and universal charts, respectively. Universal charts consist of an if-scenario (prechart) whose completion leads to the execution of a then-scenario (mainchart), whereas existential charts represent sample runs that must occur at least once. An LSC specification $S$ is defined as the disjoint union of a set of universal charts $S_U$ and a set of existential charts $S_E$.

LSCs provide various modeling constructs like subcharts, branches, and loops which can be used to specify complex behavior within a scenario as well as so-called forbidden elements to specify events that are not allowed to occur or conditions that are not allowed to hold. An LSC $L$ is defined to be

$$L = \langle I_L, V_L, M_L, [Pch_L], A_L, C_L, SUB_L, ITE_L, LOOP_L, M_L, C_L, temp \rangle$$

where $I_L$ is the set of LSC instances, $V_L$ is the set of variables, $M_L$ is the set of messages, $Pch_L$ is the prechart of $L$ (in universal charts), $A_L$ is the set of assignments, $C_L$ is the set of conditions, $SUB_L$ is the set of subcharts, $ITE_L$ is the set of if-then(-else) constructs, $LOOP_L$ is the set of loops, $M_L$ is the set of forbidden messages, and $C_L$ is the set of forbidden conditions in $L$. The $temp$-function assigns temperatures (hot/cold) to some of the LSC constructs like (forbidden) messages and conditions to differentiate between possible and mandatory behavior on the level of basic modeling elements.
Every instance line contains a set of locations. An instance progresses from one location to the next by participating in some activity (e.g. sending or receiving of a message) associated with the location.

Figure 4 shows an example universal LSC taken from a traffic light case study: The traffic of the crossing “Hamburger Straße/Rebenring” is controlled by 17 traffic lights ($A_1 - A_7$ for traffic lanes, $S_1$ and $S_2$ for trams, and $F_1 - F_8$ for pedestrians). Figure 4 depicts a certain phase affecting the states of some of the traffic lights.

A GUI representing the corresponding crossing is illustrated in Figure 5. The GUI is built with Altia Design, a commercial front end tool. Altia Design offers a number of enhanced features to easily build high quality GUIs without much programming effort.

A specification of behavioral requirements in the language of LSCs can be executed by the Play-Engine (see Appendix A.1). The Play-Engine enables the user to specify behavior (‘play-in’) by simply operating a GUI that represents the system under development. While the user defines the required behavior by clicking buttons or changing

---

1 located in Braunschweig, Germany
2 http://www.altia.com
object properties, the system automatically constructs the corresponding LSC.

The main contribution of the play-in/play-out approach is that the ‘played-in’ LSCs can be directly executed (‘played-out’) by the PLAY-ENGINE without implementing any additional code. The underlying play-out algorithm allows the user to operate the GUI as if it would be the final system. During play-out, the PLAY-ENGINE monitors the sequences of user and system actions, and reacts on each input from the user or the environment according to the specified behavior. The reactions result in changes of the GUI application like changing the state of LEDs (e.g. of the traffics lights) or displaying values. Play-out can be used to simulate the requirements specification as an executable prototype of the application as well as to test and validate it. For details on LSCs and the play-in/play-out approach, we refer to [13] and [14].

The traffic light case study also includes adaptive behavior: The duration of each traffic light phase can be adjusted by executing a MAPE cycle, thus optimizing the traffic flow. For this purpose LSCs were introduced specifying the monitoring, analyzing, planning, and executing phase, respectively.

- Monitoring LSC: Collects information about traffic flow since the last cycle.
- Analyzing LSC: Calculates the length of the average queue of each traffic light phase.
- Planning LSC: Chooses a strategy for optimizing the traffic flow by adjusting the duration of the traffic light phases.
- Executing LSC: Adjusts the duration of the traffic light phases.

The Play-Engine has turned out to be adequate for parameter adaption as demonstrated by the case study. Furthermore, LSCs constituting ‘anti-scenarios’ have been used for guaranteeing safety properties: Pairs of traffic lights that are not allowed to be simultaneously in a not-red state are monitored and violations of this rule are indicated.

Figure 5: Prototypical GUI of a set of traffic lights at the crossing “Hamburger Straße/Rebenring”
3.2. Statecharts

Statecharts were initially introduced by [15] as an extension of finite state machines and have become a part of the OMG UML specification in the meantime [29]. We refer to the CASE tool Rhapsody [18] and its statechart semantics [11] as the underlying implementation language. Statecharts take an intra-object, state-based approach characterizing the system’s behavior by a collection of communicating statecharts, each one representing an object. With statecharts, the conventional formalism of state machines is essentially extended by hierarchy, concurrency, complex transitions by expressive, implementation-oriented ECA-rules, and communication, allowing different parts of the system to interact with each other.

Formally, statecharts are a collection of state nodes, hierarchically related, and connected by edges. A statechart $SC$ is defined to be

$$SC = \langle S_{SC}, T_{SC}, E_{SC}, COND_{SC}, ACT_{SC} \rangle$$

where $S_{SC}$ is the set of states, and $T_{SC}$ is the set of transitions. A transition relation $t \in T_{SC}$ might be labeled with an event expression $e \in E_{SC}$, condition expressions $c \in COND_{SC}$, and action expressions $a \in ACT_{SC}$. Hierarchical relations are specified by nested XORStates, while ANDStates divided into several parallel regions express concurrency. In Rhapsody, the behavior of a system is a set of possible runs consisting of a series of detailed snapshots of the system’s situation (status). The status is changed by executing a step composed of microsteps. As a response to an occurrence, the system undergoes a series of microsteps as part of the run-to-completion principle, until it reaches a final status. For a detailed description of Rhapsody’s step-semantics, we refer to [11].

Figure 6 shows a statechart of a MAPE cycle which is part of a statechart model realizing the adaptive traffic light control as described in the previous subsection. The functionality of each part of the MAPE cycle corresponds to the behavior specified by the LSCs. The statechart of the MAPE cycle belongs to a self manager class which is responsible for parameter adaption of the traffic light phases.

![Figure 6: Statechart of a MAPE cycle](image)
For each part of the MAPE cycle, a corresponding statechart exists as substate of the states of the MAPE cycle (see Figure 6). The statechart of the monitoring part is depicted in Figure 7. For each of the four traffic light phases, information about the queue length is collected for computing the average queue length.

Statecharts and their ability of using hierarchy are a suitable means for modeling the traffic light system. In case of the traffic light system, one benefits from the state-based view on the system: (1) Each traffic light is in one of four phases (states): Red, amber, green, red-amber. (2) The self manager is in one of four states: Monitoring, analyzing, planning, executing. (3) The traffic light system handles four traffic situations (phases), where some traffic lights are green and some are red, respectively. Transitions between traffic situations result in amber/red-amber states of the corresponding traffic lights.

Thus, building a statechart model of the traffic light system including the self manager is rather trivial. The requirements level model covers the functional behavior of the system completely but does not take communication between the distributed parts of the traffic light system into account. Thus, transition from requirement to design requires further information like communication between the controller and the traffic lights, and has also to deal with aspects like loss of information.

### 3.3. Activity Diagrams

In industrial software engineering, activity diagrams are the natural choice for the modeling of workflows, business processes, and web-services. UML activity diagrams are very suitable for specifying use cases in the early phase of requirements specification as they provide a high-level, inter-object view on the system under development. Since the new version 2 of the Unified Modeling Language (UML), expressiveness of activ-
ity diagrams has increased significantly. In UML 2, activity diagrams are based on a completely redesigned meta-model. Several new structuring constructs like interruptible regions (enabling hierarchy) have been introduced, leading to more concise and graspable diagrams.

Activity diagrams (see Figure 8) are described as a directed graph of nodes and edges. ActivityNodes can belong to some ActivityGroup like interruptible activity regions which may be nested. ActionNodes that are specializations of ActivityNodes can be labeled by ActivityPartitions. An ActivityPartition is a kind of activity group for identifying actions that have some characteristics in common: ActivityPartition may be entities of the system being responsible for executing the appropriate actions.

Activities may invoke other activities using CallBehaviorActions where the call of an activity is indicated by placing a rake-style symbol within the node. AcceptEventActions are actions waiting for an event triggered by SendSignalActions. A wait time action is realized by AcceptTimeEventActions, a specialization of AcceptEventActions.

ControlNodes are widely known from the previous versions of the UML. As activities can contain concurrent token-flow at runtime, a single flow can be terminated using FlowFinalNodes while ActivityFinalNodes terminate all flows in an activity.

ActivityEdges can be labeled by guards. An important improvement is the introduction of InterruptibleActivityRegions for terminating the token flow in a portion of an activity by traversing InterruptingEdges. Actions can be labeled by local pre- and postconditions which should hold when execution starts and completes, respectively.

Formally, an activity \( \alpha \in \mathcal{A} \) is defined as a tuple:

\[
\alpha = \langle \text{name}, P, N, E, \text{IAR}, \text{in}, \text{out}, \text{source}, \text{target}, \text{guard}, \text{iar} \rangle
\]

where name is the name of the activity, \( P \) is the set of activity partitions, \( N \) is a set of
activity nodes, $E$ is a set of activity edges, $IAR$ is the set of interruptible activity regions, in : $N \rightarrow \mathcal{P}(E)$ gives the set of incoming edges of an activity node, out : $N \rightarrow \mathcal{P}(E)$ gives the set of outgoing edges of an activity node, guard : $E \rightarrow B$ gives a boolean expression $b \in B$ as a guard on an edge, source : $E \rightarrow N$ gives the source node of an activity edge, target : $E \rightarrow N$ gives the target node of an activity edge, and iar : $N \alpha \cup IAR \alpha \rightarrow IAR \alpha \cup \{\emptyset\}$ returns for each construct the interruptible activity region which it is directly surrounded by.

The set of activity nodes $N$ of an activity is a union of disjoin sets: $N = N^{init} \cup N^{afinal} \cup N^{final} \cup N^{dec} \cup N^{merge} \cup N^{fork} \cup N^{join} \cup ACT \cup N^{o}$ where $N^{init}$ is a set of initial nodes, $N^{afinal}$ is a set of activity final nodes, $N^{ffinal}$ is a set of flow final nodes, $N^{dec}$ is a set of decision nodes, $N^{merge}$ is a set of merge nodes, $N^{fork}$ is a set of fork nodes, $N^{join}$ is a set of join nodes, $ACT$ is a set of action nodes, and $N^{o}$ is a set of object nodes. The set of action nodes $ACT$ of an activity is a union of disjoin sets: $ACT = ACT^{b} \cup ACT^{cb} \cup ACT^{ss} \cup ACT^{ae} \cup ACT^{at}$ where $ACT^{b}$ is a set of action nodes, $ACT^{cb}$ is a set of call behavior action nodes, $ACT^{ss}$ is a set of send signal action nodes, $ACT^{ae}$ is a set of accept event action nodes, and $ACT^{at}$ is a set of accept time event action nodes.

**Live Activity Diagrams**

Unfortunately, UML has yet again failed to define a formal semantics, as would be necessary to take full advantage of the UML, e.g., in automated tools. Thus, inspired by the language of Live Sequence Charts [5] and the Play-Engine tool [13], Live Activity Diagrams (LAD) where proposed in [26] based on an executable semantics of activity diagrams. In the following, the main contributions of the LAD approach are explained:

**Simulation** LADs provide a executable step semantics and aim to support the early phase of requirements specification by directly executable models: The behavior of the system is specified by a set of LADs that refer to an underlying class- and object system constituting the structure of the system under development. A tool was implemented - based on the eclipse plugin framework - as an execution engine for a given set of LADs and an underlying class- and object system [27]. The user can operate the system by inducing sequences of events and observing the systems responses.

**Validation** LADs not only enable the specification of user requirements but also the specification of safety properties, i.e. properties, that must not be violated at runtime. Thus, the syntax definition of UML 2 activity diagrams is extended by a temp-function assigning a temperature - hot or cold to action nodes and to local pre- and post conditions. A hot action node must be executed completely and a hot local pre- and post condition must not be evaluated to false, otherwise, the requirements are violated.

**Real-time** To deal with timing properties, a single clock object with one property, time, is introduced. The time property can be referred to in guard conditions and local
Figure 9: LAD model for Lego Mindstorms NXT

pre- and postconditions to handle a rich set of timing constraints and time-based behavior. Furthermore, activity diagrams provide a timer construct - accept time event actions - that can trigger events after particular time intervals.

Dynamic activity execution At runtime, multiple activations of an activity may be active concurrently using different copies of activities. In each copy, the activity partitions may be bound to different objects, and local variables may refer to different values.

Figure 9 gives an example of an LAD model which refers to the Lego Mindstorms NXT platform and the leJOS Virtual Machine [1], a tiny Java Virtual Machine ported to the LEGO NXT brick. The underlying class- and object system is depicted on the left side in Figure 9, including classes that represent input devices (Button, UltrasonicSensor),

3see http://lejos.sourceforge.net
output devices (Motor, Sound, LCD), and the NXT control unit. On the right side, two activities CheckDistance and FindObstacle are shown, that specify the following behavior:

Whenever the enter button is pressed, the motor shall start moving forward and check the distance to an obstacle, which is placed en route. The distance shall be checked every millisecond. If the distance is smaller that 20 cm, the motor shall stop moving and a beep alert shall be issued. Additionally, the two operation modes - approaching to the obstacle and falling below a predefined distance to the obstacle - are displayed via the LCD of the brick.

**Tool-supported Simulation and Validation**

The model depicted in Figure 9 can be specified and simulated using the LAD eclipse framework [27] which has proven a powerful tool for simulating and validating functional requirements in multiple case studies (see e.g. [27, 25]). A validated set of LADs can then serve as a basis for the implementation phase.

During the simulation, the LAD model is stimulated by a sequence of user interactions: The model reacts on external stimuli like user inputs (e.g. by clicking buttons), and inputs from the external environment of the system (e.g. sensor signals). External stimuli are regarded as events which can be received by accept event actions: Pushing the enter button for instance can be realized by an event buttonPressedSignal() for which an appropriate accept event action exists (see Figure 9). The execution engine provides a dialog to enter input stimuli.

The simulation framework depicts the ongoing reactions of the system during execution: The execution of an action may force the corresponding object of the underlying object system to change its state by modifying the object’s properties. Besides, the execution steps can be traced as they are shown textually. A visualization of the executed activities is still under development.

LADs in conjunction with the simulation platform support the early phase of requirements specification by directly executable models. However, the relation between a given set of LADs constituting inter-object behavior descriptions and an appropriate implementation remains rather vague. As a first step towards implementation, we have developed a plugin as an extension of the LAD eclipse framework by enabling hardware connection: A real Lego Mindstorms NXT robot is connected to the execution framework via Bluetooth [34]. During the simulation of the model, commands like forward(), stop(), and beep() are transmitted to the NXT, whenever an action node containing the appropriate command is entered. After receiving a command, the client software on the NXT calls the corresponding method of the leJOS API thus controlling the behavior of the robot. In the opposite direction, sensor values and button signals are transmitted continuously to the execution framework running on a PC. The pushing of the enter button on the NXT e.g. issues the execution of the two activities depicted in Figure 9. In addition, the distance value is continuously transmitted to the simulation engine and checked within activity CheckDistance.
Furthermore, we have implemented a connection between the LAD execution framework and Repast Simphony, thus supporting agent-based modeling and simulation by behavioral specifications based on executable activity diagram models (see Section 4).

**Case Study “Line Follower”**

We have evaluated our approach including the extension of the LAD simulation framework by an example scenario “Line Follower” which is well known in the mobile robotics domain. As an advanced feature, two robots shall follow a black line and keep a predefined distance. The line is detected by a light sensor and the distance is measured by an ultrasonic sensor (see Figure 10).

The LAD model specifying the functional behavior contains 6 activities: **Calibration**, **Move**, **CheckLine**, **CheckDistance**, **FindLine**, and **Wait** [34]. The diagrams consist of an average of 6 action nodes. Accept event actions play an important role for receiving signals from the robot, accept time event actions for specifying time-dependent behavior, and interruptible activity regions for terminating the execution of a certain behavior.

The model can then be simulated, and thus “driving” the robots that continuously send the measured values of the distance- and light sensor to the execution environment. Hence, the modeler can instantaneously perceive if the model meets his/her demands by direct simulation. The execution of an activity can be triggered by the reception of a signal, e.g. `buttonPressedSignal()`. As activities can be activated multiple times i.e. multiple activations of an activity are executed concurrently, we handle each activation using different “copies” of the activity [27].

At runtime, the interplay of activities/copies may result in unexpected or undesired
behavior. The visible behavior of the system at runtime may not uncover the violation of a safety property. To facilitate the detection of modeling errors and specification violations, LADs provide hot/cold modalities: Safety properties - properties that must be fulfilled at runtime - are specified by hot local pre- and postconditions on action nodes. Violations of such conditions at runtime indicate violations of the requirements specification: e.g. if the distance between the two robots is lower than the predefined value. Furthermore, hot false local pre- and postconditions can be used to specify forbidden behavior. If a forbidden workflow occurs at runtime so that a hot false local pre- or postcondition is reached, a violation is indicated. Thus, for the purpose of validation of the model, the existing activities are extended by hot elements, and/or additional activities for monitoring the execution at runtime are specified. Besides, actions can be hot denoting mandatory behavior: Hot actions must be executed completely otherwise the requirements are violated.

In the line follower case study, multiple hot elements have been added in nearly every activity. The class- and object system has been extended by a class RuntimeMonitor containing a set of properties that are checked by hot local pre- and postconditions. In the line follower case study, some functions have precedence over others and some functions must not be executed all at the same time. Thus, the class RuntimeMonitor provides a property reflecting the state (active, inactive) of each function. The state is set within the 6 activities specifying the behavior of the different functions.

Compared to sequence diagrams or LSCs, the gap between an LAD requirement specification and an intra-object state-based model is much smaller. Thus, an applicable automatic code generation or model transformation comes into reach. In contrast to statecharts [15], activity diagrams specify the workflow of a certain feature instead of the states, in which parts of the system may reside during execution. Due to this kind of view, activity diagrams are closely related to use cases and are more suitable to specify the systems requirements of a reactive system during the analysis phase. A power window, as an example of a reactive system, typically comes with a number of features like automatic open/close, child lock, manual open/close, finger protection, etc. When people describe the behavior of such a systems, they won’t say things like “The power window can be in ignition-turned-on-mode or in ignition-turned-off-mode; in the first case, the system can react on input stimuli . . . by the following reactions . . . ; in the second case . . . .” Rather, you find people saying things like “If I turn on the ignition, and then press the power window button in closing direction, the window shall start closing until it reaches the final position or until I release the button.” Thus, in many cases, it is a lot more natural to describe the reactive behavior of a system by workflows via activity diagrams than by state-based models.

LAD have proven a clear and expressive specification formalism, combining the advantages of an interaction and a control oriented view. An extension of the model concerning adaption by a self manager has turned out to be straightforward.
3.4. PobSAM

Due to the fact that self-adaptive systems are often complex systems with greater degree of autonomy, it is more difficult to ensure that a self-adaptive system behaves correctly. Hence, one of the main concerns to developing self-adaptive systems is providing mechanisms to trust whether the system is operating correctly where formal methods can play a key role. *Flexibility* is another main concern to achieve adaptation in software systems. Since, hard-coded mechanisms make tuning and adapting of long-run systems complicated, so we need methods for developing adaptive systems that provide a high degree of flexibility.

Recently, the use of policies has been given attention as a rich and abstract mechanism to achieve flexibility in the self-managing systems. Policies allow us to separate the rules that govern the behavioral choices of a system from the system functionality giving us a higher level of abstraction; so we can change the system behavior without changing the code or functionality of the system. PobSAM (Policy-based Self-Adaptive Model) [23] is a flexible formal model for developing, specifying and verifying self-adaptive systems that uses policies as the principal paradigm to govern and adapt the system behavior. A PobSAM model is the composition of three layers:

- **Managed Actors Layer** This layer is dedicated to the functional behavior of a system and contains computational actors. Actors are governed by autonomous managers using policies to achieve predefined goals.

- **Autonomous Managers Layer** The main layer of PobSAM is the autonomous managers layer containing the autonomous managers. Autonomous managers are meta-actors responsible for monitoring and handling events by enforcing suitable policies. A manager may have different configurations and dynamic adaptation is performed by switching among those configurations. Each configuration consists of two classes of policies: governing policies and adaptation policies. Using governing policies, the manager directs the behavior of managed actors by sending messages to them. Adaptation policies are used to switch between configurations to adapt the system behavior properly. It is possible to define configurations, governing policies and adaptation policies dynamically which is a major advantage for developing evolving and complex systems. Thus, we can change the system behavior as well as adaptation schemes by changing policies without changing the application code.

- **View Layer** In PobSAM, each actor provides its required state information to the relevant managers. Not all aspects of the operational environment have direct influence on the behavior of managers. The views provide only the required information for managers. The view layer is composed of views that provide a view or an abstraction of an actor’s state that is adequate for the managers’ needs.

PobSAM has a formal foundation that employs an integration of algebraic formalisms and actor-based models. Let $\Pi = \langle M, V, A \rangle$ denote a PobSAM model which M, V and A
represent the set of managers, views and actors respectively. The encapsulation of state and computation, and the asynchronous communication make actors a natural way to model distributed systems. Thus, an actor-based model named Rebeca [37] is used to specify the computational environment of a self-adaptive system, i.e. $A$ is expressed as a Rebeca model. However, it is worth noting that this formalism can be replaced by other asynchronous modeling languages. A manager such as $m \in M$ is defined as the pair $m = (C, c_{\text{init}})$ where $C = \{c_1, ..., c_n\}$ and $c_{\text{init}}$ indicate the set of $m$’s configurations and the initial configuration of $m$, respectively. A simple configuration $c_s$ is defined as $c_s \triangleright (g, p)$ where $g$ and $p$ indicate the governing policy set and the adaptation policy of $c_s$, respectively.

**Governing Policies** A simple governing policy $g_s = [o, \varepsilon, \psi, a]$ consists of priority $o \in N$, event $\varepsilon$, optional condition $\psi$ and action $a$. Events are defined as execution of a message server, sending a message to an actor, creating new actor or holding a specific condition in the system. Actions can be composite or simple. A simple action is in form of $r.\ell(v)$ which denotes the message $\ell(v)$ is sent to the actor $r$. Composite actions are created by composing simple actions using sequential composition($\rightarrow$), parallel composition($\parallel$), non-deterministic choice($+$) and conditional choice operators as follows. Moreover, the special constants $\delta$ and $\epsilon$ are the deadlock and empty actions:

$$a \triangleq a \rightarrow a | a \parallel a \ | a + a \ | [\psi:a : a] \ | r.\ell(v) \ | \delta \ | \epsilon$$

Whenever a manager receives an event, it identifies all the governing policies that are triggered by that event. For each of the activated policies, if the policy condition evaluates to true, its action is requested to execute by instructing the relevant actors to perform actions through sending asynchronous messages. Governing policies are expressed using the set algebra as follows:

$$g, g' \triangleq g \cup g' \ | g - g' \ | g \cap g' \ | \{g_s\} \ | \emptyset$$

**Adaptation Policies** Whenever an event which requires adaptation occurs, relevant managers are informed. However, the adaptation cannot be done immediately and when the system reaches a safe state, the manager switches to the new configuration. Therefore, a new mode of operation called adaptation mode is introduced in which a manager runs before switching to the next configuration. This feature allows us to guide the adaptation process safely. There are two kinds of adaptations called loose adaptation and strict adaptation. Under loose adaptation, the manager enforces old policies while in the strict adaptation all the events are ignored until the system passes the adaptation mode and reaches a safe state. Adaptation policies are defined using an algebraic language as follows:

$$p \triangleright= [c]_{\xi, \psi, \gamma, \lambda, \delta} \ | p \oplus p \ | \eta$$

$$c \triangleq [\psi:c : c'] \ | c \Box c' \ | c_s \ | A$$
in which \( c, \xi, \varphi, \gamma, \lambda \) and \( \vartheta \), respectively denote an arbitrary configuration, an event, the conditions of triggering adaptation, the conditions of applying adaptation, adaptation type (loose or strict) and the priority of adaptation policy. Values \( \top \) and \( \bot \) of \( \lambda \) denote strict and loose adaptations, respectively. Adaptation policies of a manager are defined as composition(\( \oplus \)) of the simple adaptation policies. Furthermore, \( \eta \) indicates the null adaptation policy. Terms \( \psi?c : c' \) and \( c \sqsubseteq c' \) represent the conditional and non-deterministic choices of two configuration terms while \( c_s \) and \( \Lambda \) indicate simple and empty configurations, respectively.

Informally, the simple adaptation policy \( [c]_{\xi, \varphi, \gamma, \lambda, \vartheta} \) means when the event \( \xi \) occurs and the triggering condition \( \varphi \) holds, if there is no other triggered adaptation policy with the priority higher than \( \vartheta \), the manager evolves to the strict or loose adaptation modes based on the value of \( \lambda \). When the condition of applying adaptation \( \gamma \) becomes true, it will perform adaptation and switch to the next configuration.

Operational semantics of PobSAM is described with prioritized conditional state transition systems. The formal foundation, the modular model, and separation of adaptation concerns are helpful to develop rigorous analysis techniques for PobSAM. A behavioral equivalence theory is provided to reason about PobSAM models based on the notion of prioritized splitting bisimulation in [22]. The sound and complete axiomatizations is presented for this kind of bisimulation for policy actions, governing policies, adaptation policies, configurations and managers. Thus, we can reason about policy actions, governing policies, adaptation policies, configurations and managers compositionally without need to construct the whole model of the system. This is a main advantage for evolving systems whose policies and configurations change dynamically. Moreover, the distinction between the underlying computational environment and the required state information of actors makes analyzing managers much simpler, i.e. given the view layer of a PobSAM model, it is possible to analyze the manager layer independently from the actor layer which can decrease the complexity of verification to a reasonable extent.

As policies direct and adapt the behavior of a policy-based self-adaptive system, thus it is required to understand and control the overall effect of policies on the system behavior. A classification of possible conflicts between policies is proposed in [24] and the conflicts are formalized as temporal specifications.

Furthermore, PobSAM is a modular model which can support both behavioral and structural adaptations. At present, we are extending PobSAM to support structural adaptations as well as behavioral adaptation. Structural adaptation of a PobSAM models is performed by removing/adding actors and managers from/to the model dynamically. Thus, we can model changing the structure of the system as well as its behavior.

4. Case Study

We have applied activity diagrams on the modeling of an airport departure scenario, which is part of a fictional scenario of a smart airport as described in [6]. On the basis of the smart airport scenario, the research questions addressed by different research
projects of the NTH Focused Research School for IT Ecosystems are motivated [6]. In the future a joint demonstrator of an IT ecosystem, which supports this scenario will be provided.

The scenario describes an exemplary sequence of events on a usual day at an airport like Frankfurt Airport [8]. We assume, that an IT ecosystem is established at this airport, consisting of several IT components and subsystems. We will accompany Bob, Anna, and Chris during a travel to show the benefits, they would gain from an IT ecosystem. In the scenario the protagonists Bob, Anna, and Chris use small devices called SmartFolk. SmartFolk can be imagined as devices with some computing power like PDAs. The SmartFolk themselves represent their owners within the IT ecosystem and act as an interface to the IT ecosystem. Additionally, there are observation systems (e.g. SmartCameras) placed around the area, which gather and provide information (e.g. the current traffic volume) and possibly changing requirements or arising disturbances.

The complete scenario as described in [6] consists of 17 scenario steps. Within the AIM project, an adapted version of the smart airport scenario is regarded which focuses on autonomous transport of the passenger and their baggage before the departure. Next, we cite parts of the scenario (as completely specified in [6]) constituting the basis of the AIM airport departure scenario.

The AIM scenario focus on two scenario steps of [6]: “Transportation Request” and “Baggage Claim”. Figure 11 gives an overview on the use cases of the these scenario steps.

![Use case diagram “Airport Departure”](image)

**Figure 11: Use case diagram “Airport Departure”**

**Step 5** (Transportation Request). At an entrance of the airport, Anna requests transportation using her SmartFolk and waits for an autonomous transportation vehicle (ATV), to bring her to the designated check-in desk. However, at the same time, several
large groups of travelers arrive at the train and bus station near Anna’s entrance and are moving towards her position. She does not know, that at this moment, most ATVs are at a location far away from this entrance, and, by coincidence, the majority also reports a low battery power level and needs to visit a recharge station soon. Noticing the growing crowd of travelers at her location, Anna is surprised that after a short while, a sufficient number of ATVs is arriving to cope with the waiting passengers.

**Step 9 (Baggage Claim).** Going away from the check-in desk Anna asks herself how her baggage is now transported over the airport. The transportation of baggage is done by an autonomous transportation service. A variety of ATVs of different sizes performs this task by self-organization. The baggage items must be carried between different locations in the airport like check-in desks, baggage security check stations, start and landing zones of airplanes, etc. Additionally, there are observation systems (e.g. cameras, sensors, RFID readers) placed around the area, which gather and provide information (e.g. the current traffic volume) and possibly changing requirements or arising disturbances. This information is used by the ATVs (in terms of self-organization and interaction) in order to achieve a good performance of transportation.

The airport departure scenario refers to a simplified map of the airport (see Figure 12). The scenario-related places are: Entrances, check-Ins, gates, baggage claims, parking positions, charging stations, and roads (main roads and side roads).

Agents of the scenario can be identified as passengers, ATVs, SmartFolks, and SmartCameras.

Based on this world model, we have modeled the behavior of the airport departure scenario using activity diagrams. The scenario-related research questions can be characterized as follows: Are activity diagrams adequate to model the reactive behavior of the agents? How can safety properties be assured? Can dynamic aspects like the unlimited number of transport requests be handled? To which extend can adaptive behavior be specified? Is the agent architecture as proposed in [17] suitable for modeling the communicating agents of this scenario?

Figure 13 shows the control cycle of the robot: Transport requests are continuously received via an accept event action. Each transport request is handled using a call behavior action for invoking the corresponding activity for executing the request. In case of a low battery, the robot has to move to a charging stating which is modeled by a separate activity diagram. Here, we use a synchronous call for invoking activity “Charge” as the ongoing flow has to be blocked until the status of the battery is OK. In contrast, transport requests are handled using asynchronous calls as the flow has to proceed immediately after receiving the request. Otherwise, only single requests can be served. Parameters are transmitted to the invoked behavior which are different for each request.

An instance of activity “Execute Request” is created for each request (see Figure 14). Activity “Execute Request” uses advanced constructs of activity diagrams like interruptible activity regions for enabling hierarchy, concurrent flows, and diagram connectors to improve the readability of the diagram. We use broadcast signal actions for communication with other agents. The notion of broadcast signal actions is proposed by [33] as UML does not provide a notion for broadcast signal actions. Signal actions in gen-
eral are used for interaction between the agent and its environment (including other agents), and between different parts/layers of the agent. As explained in Section 3.3, action execution may refer to an underlying class- and object system representing the structure of the system under development. Activity diagrams can be used to define the behavior of each component of a given architecture of an adaptive system. By this intra-object-view, each action of an activity refers to the same component of the agent. In addition, activity diagrams also enable inter-object behavior descriptions which cut across the boundaries of the system’s components. An inter-object-view has turned out to be adequate for specifying the reactive behavior of the agent, whereas an intra-object specification is more suitable for modeling a closed control cycle of a self manager. The set of activities for driving the execution can be extended by activities for validating safety and liveness properties at runtime.

Figure 12: Grid of the smart airport
The LAD simulation framework has been extended by a connection to the Repast Simphony toolkit. Repast (Recursive Porous Agent Simulation Toolkit) Simphony is an open source agent-based modeling toolkit that simplifies model creation and use. Repast offers a rich variety of features, e.g., built-in simulation results logging and graphing tools, a fully concurrent multithreaded discrete event scheduler, and built-in

\[ http://repast.sourceforge.net \]
adaptive features such as genetic algorithms and regression. The connection to the Repast toolkit enables enhanced capabilities for visual execution of the scenario.

Figure 15: Connection to Repast Simphony

5. Conclusion

In recent years, adaption has been addressed by many research projects for instance from organic/autonomic computing proposing different architectures for self-organizing systems, e.g. observer/controller architectures [31], and the MAPE cycle [20]. The aim is to define a system structure for enabling an implementation of a system that adapts dynamically to the current conditions of its environment. However, the modeling of the behavior of the system’s components itself has also to be addressed. We have prototypically compared some existing modeling languages for behavioral modeling with respect to their suitability for modeling adaptive behavior based on the MAPE cycle. We have applied activity diagrams for the modeling of an airport departure scenario, which is part of a fictional scenario of a smart airport as described in [6]. Within the AIM project, an adapted version of the smart airport scenario is regarded which focuses on autonomous transport of the passengers and their baggage before the departure. However, activity diagrams need to be semantically clarified and to be evaluated concerning their expressiveness.

Compared to statecharts, activity diagrams are semantically more complex due to e.g. overlapping activity invocations, and the higher degree of concurrency which can be expressed by activity diagrams. Thus, activity diagrams have turned out to be adequate
for modeling different levels of adaptive behavior. An unlimited number of transport requests can be handled due to dynamic activity invocations. Activity diagrams combine the advantages of an interaction and a control oriented view. By our extensions concerning hot/cold modalities, mandatory and optional requirements are distinguished and liveness properties can be enforced. The tool for direct simulation of activity diagram models in conjunction with the Repast toolkit helps to identify errors and specification violations yet in early phases by appropriately tailored views for static and dynamic system properties.

We have also analyzed PobSAM (Policy-based Self-Adaptive Model) [21] as a flexible model, for developing, specifying and verifying self-adaptive systems. PobSAM has turned out to be adequate for preventing/resolving conflicts. Each agent has a set of policies to control behavior which may change their policies dynamically. In order to prevent/resolve conflicts, PobSAM can change policies (adaptation mechanism in PobSAM). For example, if resources are not enough, the agent switches to a new configurations to adapt with resource limitation. Different sets of policies are defined for different situations. Also, each policy has a priority where priorities can help to prevent/resolve conflicts, e.g. in case of resource limitation, PobSAM may handle requests with higher priorities first. Another feature of PobSAM is its capability to change the structure of the system, e.g. if resources are not sufficient, PobSAM may reconfigure the system by adding new resources.
A. Modeling Tools

A.1. Play-Engine

Website: http://www.wisdom.weizmann.ac.il/~playbook

The PLAY-ENGINE was released along with the Playbook [13] in 2003 (see Figure 16). Scenarios are ‘played in’ very intuitively by triggering the system’s GUI. In the PLAY-ENGINE the scenarios are represented as LSCs. Afterwards the user can probe the requirements specification in the play-out phase: Without further code generation, the PLAY-ENGINE interprets the LSCs and provides together with the GUI an abstract virtual prototype that can be used to simulate and validate the system’s behavior. The PLAY-ENGINE has proven a powerful tool for requirements specification and validation in a number of case studies on biological systems [7, 19] and telecommunications [4, 12].

Figure 16: PLAY-ENGINE tool
A.2. Rhapsody

Website: http://www-01.ibm.com/software/awdtools/rhapsody/

IBM’s Rhapsody is a UML/SysML-based model-driven development tool for real time and embedded systems and software. Rhapsody enables early validation of behavior of embedded systems and software by model execution. Simulation brings diagrams to life for design level debugging and early validation. Furthermore, Rhapsody enables full behavioral code generation for C, C++, Java and Ada. The Rhapsody-tool together with a simple application visualizing the result values is shown in Figure 17.

Figure 17: Rhapsody tool
A.3. LAD-Simulation

Based on the syntax and formal semantics of LADs by [25, 26] a tool has been implemented [36] in the form of an Eclipse plug-in\(^5\) supporting the simulation and validation of LAD models (see Figure 18), and the modeling of LADs by an integrated editor. When executing the model, objects of the underlying object system change their state, i.e. the corresponding object properties are modified. Although a visualization of the executed activities is still under development, the execution steps can be traced as they are shown textually. We have implemented a connection between the LAD execution environment and Repast Simphony, a widely used, free, and open-source agent-based modeling and simulation toolkit. Thus, the LAD execution engine supports agent-based modeling and simulation by the specification of the agent behavior via activity diagrams.

---

\(^5\)http://www.eclipse.org
References


