Helena@Work:
Modeling the Science Cloud Platform

Annabelle Klarl, Philip Mayer, and Rolf Hennicker

Ludwig-Maximilians-Universität München, Germany[**]

Abstract. Exploiting global interconnectedness in distributed systems, we want autonomic components to form teams to collaborate for some global goal. These teams have to cope with heterogeneity of participants, dynamic composition, and adaptation. Helena advocates a modeling approach centered around the notion of roles which components can adopt to take part in task-oriented teams called ensembles. By playing roles, the components dynamically change their behavior according to their responsibilities in the task. In this paper, we report on the experiences of using Helena in modeling and developing a voluntary peer-2-peer cloud computing platform. We found that the design with roles and ensembles provides a reasonable abstraction of our case study. The model is easy to understand and helps to identify and eliminate communication errors during analysis due to its solid semantic foundation. Encapsulation of responsibilities in roles makes the implementation straightforward. Although some extensions have to be added to make the realization more robust, we believe that the Helena models are a useful abstraction of concurrent ensembles.

1 Introduction

The development of distributed software systems, i.e. systems in which individual parts run on different machines connected via some sort of communication network, has always been a challenge for software engineers. Special care has to be taken to the unique requirements concerning concurrency and sharing of responsibilities. In this area, difficult issues arise particularly in those systems in which the individual distributed software components have a certain degree of autonomy and interact in a non-centralized and non-trivial manner.

Such systems are investigated in the EU project ASCENS [1], where the individual distributed artifacts are components which provide the basic capabilities for collaborating teams. These components dynamically form *ensembles* to perform some global task. We believe that the execution and interaction of entities in such ensembles is best described by what we call *roles*. They are an abstraction of the part an individual component plays in a collaboration. We claim that separating the behavior of components into individual roles leads to an easier understanding, modeling, and programming of ensemble-based systems.

Our modeling approach Helena [9,12] thus extends existing component-based software engineering methods [18] by modeling roles. Each role (more precisely role type) adds particular capabilities to the basic functionalities of a component which are only relevant when performing the role. Exploiting these role-specific capabilities, we specify *role behaviors* which the component dynamically adopts when taking over a role. For the specification of role behaviors we extend [9] by introducing a process language which allows to describe dynamic creation of role instances on selected component instances. The structural characteristics of collaborations are defined in *ensemble structures* capturing the contributing role types and potential interactions.

In this paper, we report on the experiences of using Helena in modeling and developing a larger software system. As our case study we have selected the *Science Cloud Platform (SCP)* [14] which is one of the three case studies used in the ASCENS project. The SCP is, in a nutshell, a platform of distributed, voluntarily provided computing nodes. The nodes interact in a peer-to-peer manner to execute, keep alive, and allow use of user-defined software applications. The goal of applying Helena to the SCP is to find a reasonable abstraction that serves as clear documentation, analysis model, and guideline for the implementation. We experienced that the Helena model helps to rigorously describe the concepts of the SCP. During analysis of the models, communication errors can be eliminated at early stages. As we shall discuss, the implementation also benefits from the encapsulation in roles. However, during implementation some additional effort is required to provide an infrastructure for using role abstractions on top of the component-based system. Lastly, special care has to be taken to make the system robust against communication failures and to provide communication facilities between ensembles and the outside world which is not yet tackled in Helena.

In the following sections, we first describe the case study in Sec. 2. Afterwards, we summarize the Helena modeling approach in Sec. 3 and apply it to the case study in Sec. 4. Sec. 5 describes the realization of the Helena model on the infrastructure of the SCP. Lastly, we report on experiences, give an outlook, and compare our approach to related work in Sec. 6.

2 Case Study

One of the three case studies in the ASCENS project is the *Science Cloud Platform (SCP)* [14]. The SCP employs a network of distributed, voluntarily provided computing nodes, in which users can deploy user-defined software applications. To achieve this functionality, the SCP reuses ideas from three usually separate computing paradigms: cloud computing, voluntary computing, and peer-to-peer computing. In a nutshell, the SCP implements a platform-as-a-service in which individual, voluntarily provided computing nodes interact using a peer-to-peer protocol to deploy, execute, and allow usage of user-defined applications. The SCP takes care to satisfy the requirements of the applications, keeps them running even if nodes leave the system, and provides access to the deployed appli-

cations. For a full description of the SCP, we refer to [14]. In the following, we only discuss those parts relevant for this paper.

The SCP is formed by a network of computers which are connected via the Internet, and on which the SCP software is installed (we call these *nodes*). The layout of an SCP node is shown in Fig. 1, along with the technologies involved. The dashed boxes are those parts contributed in the current work.
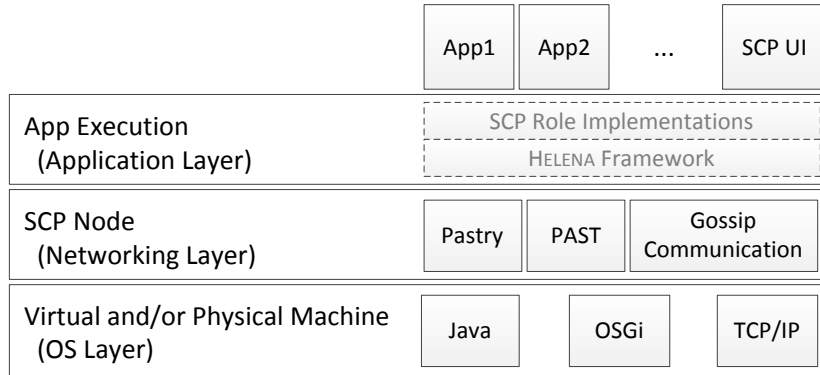

Fig. 1: SCP architecture (new parts in dashed boxes)

The bottom layer shows the infrastructure: The SCP is a Java application and thus runs in the Java VM; it also uses the OSGi component framework to dynamically deploy and run applications (as bundles). In general, plain TCP/IP networking is used to communicate between nodes on this level.

The second layer implements the basic networking logic. The SCP uses the distributed peer-to-peer overlay networking substrate Pastry [16] for communication. Pastry works similarly to a Distributed Hash Table (DHT) in that each node is represented by an ID. Node IDs are organized to form a ring along which messages can be routed to a target ID. Pastry manages joining and leaving nodes and contains various optimizations for fast routing. On top of this mechanism, the DHT PAST allows storage of data at specific IDs. On this layer, a gossip protocol [7] is used to spread information about the nodes through the network; this information includes node abilities (CPU, RAM), but also information about applications. Each node slowly builds its own picture of the network, pruning information where it becomes outdated.

The third layer (shown in grey) is presented in this paper, and implements the application execution logic based on HELENA. The dashed boxes describe the intended implementation which are discussed throughout the paper. The required functionality of the application layer is that of reliable application execution given the application requirements on the one hand and the instability of the network on the other hand. This process is envisioned as follows:

1. **Deploying and undeploying:** A user deploys an application using the SCP UI (top right). The application is assigned an ID (based on its name) and stored using the DHT (PAST) at the closest node according to the ID; this ensures that exactly one node is responsible for the application, and this node can always be retrieved based on the application name (we call this node the *app-responsible node*). If this node leaves, the next adjacent node based on ID proximity takes its place.

2. **Finding an executor:** Since each application comes with execution requirements and all nodes are heterogeneous, the app-responsible node may or may not be able to execute the application. Thus, it is tasked with *finding* an appropriate executor (based on the gossiped information).

3. **Executing:** Once an executor is found, it is asked to retrieve and run the application. Through a continuous exchange of keep-alive messages, the app-responsible node observes the executor and is thus able to select a new one if it fails. The user may interact with the application through the SCP UI.

3 Ensemble Modeling with HELENA

With HELENA, we model systems with large numbers of entities which collaborate in teams (or *ensembles*) towards a specific goal. In this section, we summarize the basic ideas and ingredients of the HELENA approach [9,12] which is centered around the notion of roles which components can adopt to form ensembles. In HELENA, we use role types to specify task-oriented capabilities and team them up for task execution in ensemble structures. Role behavior specifications describe the dynamic behavior of active instance of a role type and form an ensemble specification together with the ensemble structure. In Sect. 3.2 we extend the existing HELENA approach by a new language for specifying role behaviors in terms of process expressions which allow to describe role instance creation on certain components and role instance retrieval.

3.1 Ensemble Structures

The foundation for the aforementioned systems are components. To classify components we use *component types*. A component type defines a set of attributes (more precisely attribute types) representing basic information that is useful in all roles the component can adopt. Formally, an *attribute type* is just a named variable (for simplicity untyped) and a *component type ct* is a tuple $ct = (nm, attrs)$ such that nm is the name of the component type and $attrs$ is a set of attribute types.

For performing certain tasks, components team up in *ensembles*. Each participant in the ensemble contributes specific functionalities to the collaboration, we say, the participant plays a certain *role* in the ensemble which we classify by role types. A role type determines the types of the components that are able to adopt this role. It also defines role-specific attributes (to store data that is

only relevant for performing the role) and it defines message types for outgoing, incoming, and internal messages. Formally, a *message type* is of the form $msg = msgnm(riparams)(dataparams)$ such that $msgnm$ is the name of the message type, *riparams* is a list of typed formal parameters to pass role instances, and *dataparams* is a list of (for simplicity untyped) formal parameters for data.

Definition 1 (Role Type). *Given a set CT of component types, a* role type *rt over CT is a tuple rt $= (nm, compTypes, roleattrs, rolemsgs)$ such that*

- *nm declares the name of the role type,*
- *compTypes is a finite, non-empty set of component types (whose instances can adopt the role) with compTypes $\subseteq$ CT,*
- *roleattrs is a set of role specific attribute types,*
- *rolemsgs $= \langle rolemsgs_{out}, rolemsgs_{in}, rolemsgs_{int} \rangle$ specifies types for outgoing, incoming, and internal messages supported by the role type.*

Role types form the basic building blocks for collaboration in an ensemble. An *ensemble structure* determines the type of an ensemble that is needed to perform a certain task. It specifies which role types are needed in the collaboration and how many instances of each role type may contribute.

Definition 2 (Ensemble Structure). *Let CT be a set of component types. An* ensemble structure Σ *over CT is a pair* $\Sigma = (roleTypes, roleConstraints)$ *such that roleTypes is a set of role types over CT and for each rt $\in$ roleTypes, roleConstraints(rt) $\in$ Mult and Mult is the set of multiplicities available in UML, like 0..1, 1, $*$, 1..$*$, etc.*

For simplicity, we do not use explicit role connector types here opposed to [9] and assume that between (instances of) role types rt and rt' the messages with the same name that are output on one side and input on the other side can be exchanged. They are visualized in the graphical representation of ensemble structures (cf. Fig. 2).

3.2 Role Behavior Specifications

After having modeled the structural aspects of ensembles, we focus on the specification of behaviors for each role type of an ensemble structure. A role behavior is given by a process expression built from the null process, action prefix, nondeterministic choice, and recursion. In the following, we use X, Y for role instance variables, RT for role types, x for data variables[1], e for data expressions and ci for component instances (assuming a given repository of those); $\vec{z}$ denotes a list of z. There are five different kinds of actions. A send action is of the form $X!msgnm(\vec{Y})(\vec{e})$. It expresses that a message with name $msgnm$ and actual parameters $\vec{Y}$ and $\vec{e}$ is sent to a role instance named by variable X. The first

---

[1] We distinguish between role instance variables and data variable since role instance variables can be used as recipients for messages later on, for instance for callbacks.

parameter list $\vec{Y}$ consists of variables which name role instances to be passed to the receiver; with the second parameter list $\vec{e}$, data is passed to the receiver. A receive action is of the form $?msgnm(\vec{X} : \overrightarrow{RT})(\vec{x})$. It expresses the reception of a message with name $msgnm$. The values received on the parameters are bound to the variables $\vec{X}$ for role instances and to $\vec{x}$ for data. Internal actions are represented by $msgnm(\vec{Y})(\vec{e})$ denoting an internal computation with certain parameters. With the action $X \leftarrow \mathbf{create}(RT, ci)$ a new role instance of type RT is created, adopted by the component instance ci, and referenced by the variable X of type RT in the sequel. Similarly the action $X \leftarrow \mathbf{get}(RT, ci)$ retrieves an arbitrary existing role instance of type RT already adopted by the component instance ci. Thus, the variables $\vec{X}, \vec{x}$ used in message reception and the variable X for role instance creation and retrieval open a scope which binds the open variables with the same names in the successive process expression. The bound variables receive a type as declared by the types $\overrightarrow{RT}$.

Definition 3 (Role Behavior). *Let Σ be an ensemble structure and rt be a role type in Σ. A role behavior $RoleBeh_{rt}$ for rt is a process expression built from the following abstract syntax:*

$$
\begin{aligned}
P ::= \ & \mathbf{nil} && \text{(null process)} \\
| \ & a.P && \text{(action prefix)} \\
| \ & P_1 + P_2 && \text{(nondeterministic choice)} \\
| \ & \mu V.P && \text{(recursion)} \\
a ::= \ & X!msgnm(\vec{Y})(\vec{e}) && \text{(sending a message)} \\
| \ & ?msgnm(\vec{X} : \overrightarrow{RT})(\vec{x}) && \text{(receiving a message)} \\
| \ & msgnm(\vec{Y})(\vec{e}) && \text{(interal computation)} \\
| \ & X \leftarrow \mathbf{create}(RT, ci) && \text{(role instance creation)} \\
| \ & X \leftarrow \mathbf{get}(RT, ci) && \text{(role instance retrieval)}
\end{aligned}
$$

To be well-formed a role behavior $RoleBeh_{rt}$ must satisfy some obvious conditions: 1) For sending a message $X!msgnm(\vec{Y})(\vec{e})$ the role type rt must support the message type $msgnm(riparams)(dataparams)$ as outgoing message and the actual parameters must fit to the formal ones. Moreover, X must be a variable of some type RT which supports the same message type as incoming message. Similarly, well-formedness of incoming and internal messages is defined. 2) Role instance creation $X \leftarrow \mathbf{create}(RT, ci)$ and role instance retrieval $X \leftarrow \mathbf{get}(RT, ci)$ are well-formed if RT is a role type in Σ, and if the component instance ci if of a type whose instances can adopt a role of type RT.

Definition 4 (Ensemble specification). *An ensemble specification is a pair $EnsSpec = (\Sigma, RoleBeh)$ such that Σ is an ensemble structure, and $RoleBeh$ is a family of role behaviors $RoleBeh_{rt}$ for each role type rt occurring in Σ.*

In this paper, we do not focus on a formal semantics of ensemble specifications. However, some hints may be helpful. As a semantic basis for the evolution

of ensembles we use *ensemble automata* as defined in [9]. The states of an ensemble automaton show 1) the currently existing role instances of each role type occurring in Σ, 2) for each existing role instance, a unique component instance which currently adopts this role, 3) the data currently stored by each role instance, and 4) the current control state of each role instance showing its current progress of execution according to the specified role behavior. Ensemble automata model role instance creation as expected by introducing a fresh role instance which starts in the initial state of its associated role behavior. Retrieval of role instances delivers an existing role instance of appropriate type played by the specified component instance if there is one. Otherwise it is blocked. Concerning communication between role instances first an underlying communication paradigm must be chosen. The ensemble automata in [9] formalize synchronous communication such that sending and receiving of a message is performed simultaneously. If the recipient is not (yet) ready for reception of the message the sender is blocked. However, it is important to note that the communication style is not determined by an ensemble specification since the role behaviors specify local behaviors and thus support decentralized control which is typical for the systems under investigation. In particular, an asynchronous communication pattern can be chosen as well for the realization of an ensemble specification and this is indeed the case for the ensembles running on the SCP.

4 Modeling the SCP with Helena

Let us revisit our case study from Sec. 2 to explain the benefits of the role-based modeling approach for such a system. In the SCP, distributed computing nodes interact to execute software applications. For one app, several computing nodes need to collaborate: They have to let a user deploy the app in the system, to execute (and keep alive) the app on a node satisfying the computation requirements of the app, and to let a user request a service from the app. For each of these responsibilities we can derive a specific behavior, but at design time it is unclear which node will be assigned with which responsibility. Additionally, each node must also be able to take over the same or different responsibilities for the execution of different apps in parallel. In a standard component-based design, we would have to come up with a single component type for a computing node which is able to combine the functionalities for each responsibility in one complex behavior. Hence, the Helena modeling approach offers the possibility to model systems in terms of collaborating roles and ensembles. Firstly, roles allow to separate the definition of the capabilities and behavior required for a specific responsibility from the underlying component. Secondly, adopting different roles allows components to change their behavior on demand. Thirdly, concurrently running ensembles support the parallel execution of several tasks possibly sharing the same participants under different roles.

In the SCP, we assume given the basic infrastructure for communication between nodes (Pastry), storing data (PAST), and deploying and executing apps (OSGi) (two bottom layers in Fig. 1). We apply Helena for modeling the whole

process of application execution on top of this infrastructure. Computation nodes represent the components underlying the HELENA model.

Ensemble Structure The first step is to identify the required role types from the stated requirements in Sec. 2.

1. **Deploying and undeploying:** For this subtask, we envision two separate role types. The `Deployer` provides the interface for deploying and undeploying an app and is responsible for the selection of the app-responsible node for storing the app code. The app-responsible node adopts the `Storage` role taking care for the actual storage and deletion of the app code and initiates the execution of the app.
2. **Finding an executor:** Three further roles are required for finding the appropriate execution node. The app-responsible node in the role `Initiator` determines the actual `Executor` from a set of `PotentialExecutors` and takes care that it is kept running until the user requests to undeploy the app. A `PotentialExecutor` is a node which the `Initiator` believes is able to execute the app based on the requirements of the app. However, it might currently not be able to do so, e.g., due to its current load. The actual `Executor` is selected from the set of `PotentialExecutors` and is responsible for app execution.
3. **Executing:** Once started, the app needs to be available for user requests. The `Requester` provides the interface between the user and the `Executor` and forwards requests and responses. The `Executor` from the previous subtask gives access to the executed app.

In Fig. 2, we summarize the ensemble structure composed of these six roles graphically. The multiplicities of the role types denote that a running ensemble contains just one role instance per role type except for `PotentialExecutor` and `Requester`. Labels at the connections between roles depict which messages can be exchanged between these roles for collaboration. We explain the exchanged messages in more detail when we focus on role behaviors. For each deployed app, one instance of this ensemble structure is employed. Different components may take over the required roles in one ensemble, but a single component may also adopt different roles in the same ensemble. Moreover, different components can take part at the same time in different ensembles under different roles.

Role Behavior Specifications On the basis of this ensemble structure, we specify a behavior for each role. For the roles `Deployer` and `Storage` taking part in the first subtask, the role behaviors are rather straightforward and we give only an informal description. In the initial state the `Deployer` waits for the user to ask for app deployment and forwards the app code to the `Storage` for archiving and vice versa for undeployment. The `Storage` role starts by waiting for a request to store an app. Upon storage, it issues the creation of an `Initiator` which takes care that the app is executed. Afterwards the `Storage` is ready to provide the app code to an `Executor` or to delete it.
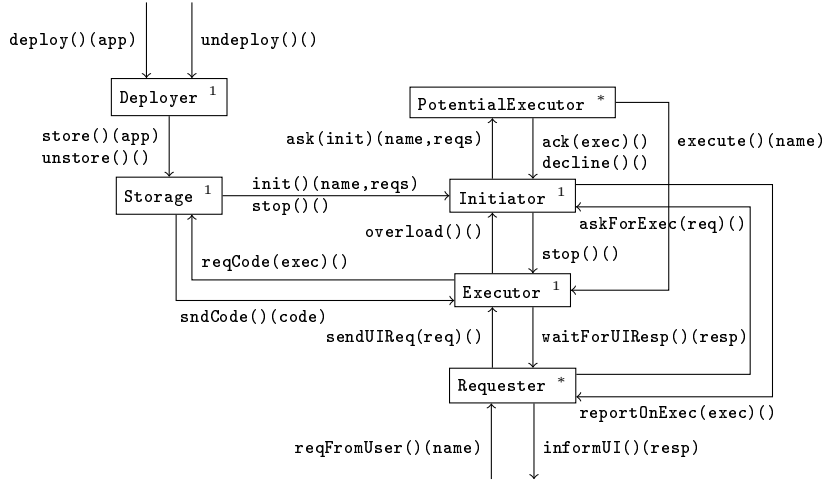
Fig. 2: Ensemble structure for app execution in the SCP

What is interesting about these two role types is which component instances are selected to adopt the roles. The `Deployer` is automatically played by the component instance where the user actually places her deployment request. When the `Deployer` creates a `Storage` it selects the component whose ID according to Pastry (cf. Sec. 2) is next to the ID of the app (given by the hash value of the app name). The uniqueness of component selection is essential since for any later communication with the `Storage`, e.g., for code retrieval, it must be possible to identify the owning component instance just from the app's name. For the same reason, we choose the owning component of the `Storage` to additionally adopt at the `Initiator` role.

The behavior of the `Requester` is also straightforward and is again informally described. In the initial state, a `Requester` waits for the user to request a service from the app. It retrieves a reference to the `Executor` from the `Initiator`[2] and forwards the request to the `Executor`. It gets back a response from the `Executor` which it routes to the user. The part played by the `Executor` in this scenario is depicted in Fig. 3c by the loop between states `e5` and `e6`.

The most interesting behavior concerns the selection of an appropriate executor. In Fig. 3, we translated the process terms of each role behavior into a labeled transition system which makes it easier to explain. The main idea is that the initiator of an app asks a set of potential executors, one after the other, for execution of the app until one of them accepts. Since each node maintains a list of all other nodes and their abilities through a gossip protocol (cf. Sec. 2), the initiator can easily prepare this list of nodes satisfying the requirements of the app based on its current belief of the network. Triggered by the reception of the

---

[2] Note that for communication with the `Initiator` its owning component must be uniquely identifiable as mentioned before.

9

init message, the Initiator starts to walk through the list. It first creates a new PotentialExecutor on the next node satisfying the requirements and asks it for execution. If it declines, the next node satisfying the requirements is asked until one accepts (states i1 to i4). As soon as a PotentialExecutor accepts, the Initiator waits for one of three messages in state i4: 1) an overload message meaning that the current Executor is not able to execute the app anymore and the Initiator has to find a new one, 2) a request for the reference to the Executor (issued by a Requester), or 3) a stop message triggering stopping the execution of the app on the Executor.

The behavior of a PotentialExecutor starts with waiting for a request for app execution. If it does not satisfy the requirements of the app (like current load), it internally decides to refuse and sends back a decline message. Otherwise, it creates a new Executor on its owner, issues the execution, and acknowledges execution to the Initiator. An Executor starts by waiting for an execute message. Then the Executor retrieves a reference to the Storage, requests and gets the app code from it and starts execution of the app (states e1 to e5). As soon as the app has been started, the Executor can answer user requests or stop execution due to internal overload or an external stop request.

Analysis The role behaviors provided by an ensemble specification can be used to analyze the dynamic behaviors of ensembles before implementing the system. A particularly important aspect concerns the avoidance of communication errors when role instances communicate. Two types of errors can be distinguished. Firstly, an instance expects the arrival of a message which never has been issued. Secondly, an instance sends a message, but the recipient is not ready to receive. Let us analyze the latter type of communication error by considering the cooperation between Initiator and PotentialExecutor. The only output action occurring in $RoleBeh_{\text{Initiator}}$ which is addressed to a PotentialExecutor is the message ask occurring in state i2. It is sent to the PotentialExecutor, named by the variable pot, which has just been created in state i1. This potential executor starts in its initial state p0 in which it is obviously ready to accept the message ask. Afterwards, the Initiator is in state i3 and is ready to receive either a decline or an ack message which both can only be sent from the PotentialExecutor. After the reception of ask the PotentialExecutor is in state p1 and it has two options: 1) It can decide to refuse the request and sends the message decline which the Initiator accepts being back in state i1. In this case, the current PotentialExecutor terminates, a new one is created, the Initiator goes to state i2, and we are in a situation which we have already analyzed. 2) The other option in state p1 is to accept the execution request, to create an Executor, to cause the Executor to start execution and then to send the message ack to the Initator who is still in state i3 and takes the message. So the instances of both roles, Initator and PotentialExecutor, work well together. Interestingly this holds whether one uses synchronous or asynchronous communication in the implementation. How such an analysis can be performed on the basis of formal verification is a challenging issue of future research.
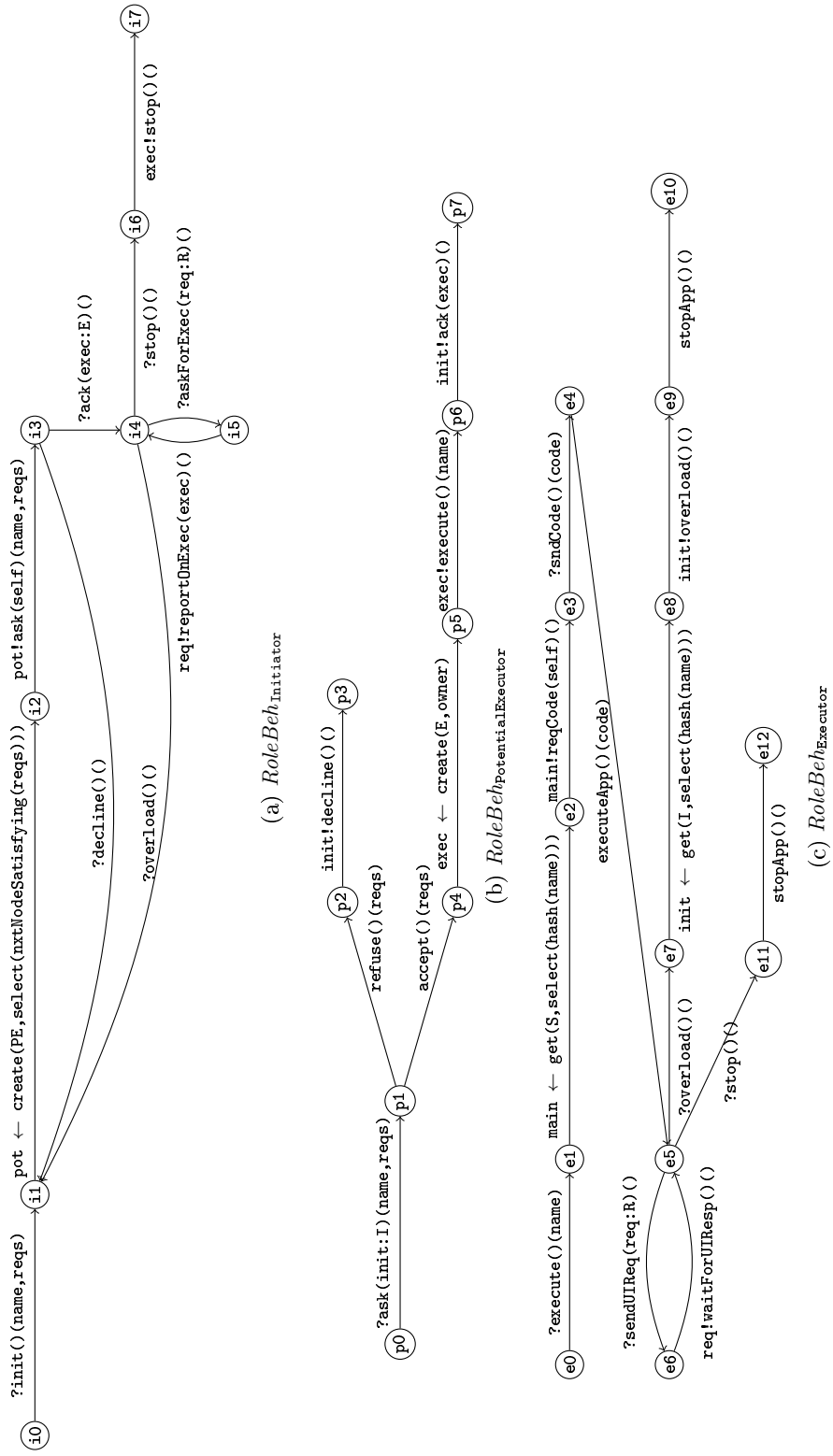
(a) $RoleBeh_{\texttt{Initiator}}$

- i0 — ?init()(name,reqs) → i1
- i1 — pot ← create(PE,select(nxtModeSatisfying(reqs))) → i2
- i2 — pot!ask(self)(name,reqs) → i3
- i3 — ?decline()() → i1
- i1 — ?overload()() → i3
- i3 — ?ack(exec::E)() → i4
- i4 — ?stop()() → i6
- i6 — exec!stop()() → i7
- i4 ⇄ i5 — ?askForExec(req:R)() / req!reportOnExec(exec)()

(b) $RoleBeh_{\texttt{PotentialExecutor}}$

- p0 — ?ask(init:I)(name,reqs) → p1
- p1 — refuse()(reqs) → p2
- p2 — init!decline()() → p3
- p1 — accept()(reqs) → p4
- p4 — exec ← create(E,owner) → p5
- p5 — exec!execute()(name) → p6
- p6 — init!ack(exec)() → p7

(c) $RoleBeh_{\texttt{Executor}}$

- e0 — ?execute()(name) → e1
- e1 — main ← get(S,select(hash(name))) → e2
- e2 — main!reqCode(self)() → e3
- e3 — ?sndCode()(code) → e4
- e4 — executeApp()(code) → e7
- e7 — ?overload()() → e5
- e5 ⇄ e6 — ?sendUIReq(req:R)() / req!waitForUIResp()()
- e7 — init ← get(I,select(hash(name))) → e8
- e8 — init!overload()() → e9
- e9 — stopApp()() → e10
- e5 — ?stop()() → e11
- e11 — stopApp()() → e12

Fig. 3: Role behaviors for Initiator, PotentialExecutor, and Executor (all role types are abbreviated)

Limitations At this point, we want to mention some restrictions underlying the current Helena approach. Firstly, we rely on binary communication and do not support broadcast yet. Though broadcast sending could be easily integrated in our process expressions, to collect corresponding answers would still be an issue. Secondly, we build ensemble specifications on a given set of components such that we cannot model situations in which components fail. However, we are aware that one of the main characteristics of our case study is that nodes may fail and leave the network at any time. We wish that such failovers are handled transparently from the role behaviors. The idea is that components are monitored such that when failing all adopted roles are transparently transferred to another component and restarted there. A further issue concerns robustness since we assume reliable network transmission in our models. We do not want to include any mechanisms for resending messages in the role behavior specifications. Like failover mechanisms, this should be handled transparently by an appropriate infrastructure.

5 Using the Helena Model for the SCP Implementation

In this section, we report on the experimental realization of the Helena model[3]. Helena separates between base components and roles running on top of them. The SCP is already built on components (the *SCP Node* layer in Fig. 1); thus, the Helena implementation can build on the given infrastructure and realize the *application layer* shown by the dashed boxes in Fig. 1 by a role-based implementation as envisioned in the Helena approach.

When modeling the application layer we leveraged the notions of ensembles, roles, and role-specific functionalities like the exchange of messages. However, a direct translation of the Helena model on the given SCP infrastructure would not be able to benefit from these abstractions and would include a lot of boiler plate code to map the role behaviors to the Pastry peer-to-peer routing substrate. We have thus opted to create a layer between the Pastry API and the role behavior implementations. This Helena *framework* amounts to around 1000 LOC and offers role-related functionality, such as the ability to create and retrieve roles via the network, and routing messages between roles. (This layer implements the same basic ideas already presented in the Helena framework [12], but is based on the SCP and thus, Pastry). In a second step, we have translated the behavioral specifications of the six roles to Java code using the previously created framework. Each of the role implementations stays below 150 LOC with another 400 LOC in message classes. In the following two subsections we discuss the framework and role implementations, respectively, stressing where direct translation of the Helena model was possible and where special care had to be taken to make the realization robust.

---

[3] The code can be retrieved from `http://svn.pst.ifi.lmu.de/trac/scp`, version v3 of the node core implementation with gossip strategy.

5.1 Implementing the HELENA Framework

A framework for implementing role behaviors needs to offer several features to role implementors.

Structural Aspects The most important concept in HELENA are *roles*. Thus, the framework must offer the ability to create role types, and to instantiate and execute them. This maps quite naturally to using one Java class per role type, and instantiating this class for role instances. A registry on each node stores all instances currently adopted by the node and allows their retrieval. To enable concurrent execution, each role instance is realized as a Java thread, running locally in the OSGi container of the current node.

The framework provides means to create, retrieve, and address existing roles on other nodes; this requires a way of addressing roles. Thus, the second important structural aspect is addressing. In Pastry, each node is already identified by a unique 160-bit identifier. It is relatively straightforward to add a similar unique identifier for roles. However, there is also another kind of structuring element which is not directly visible in the behavioral specifications: The ensemble which constitutes the environment for the roles. This can clearly be seen when looking at the functions the framework needs to offer for role handling – these are the `create` and the `get` functions. Both require knowledge about which ensemble is addressed for creating a new role or where to look for an existing role. We have thus three identifiers in use in the HELENA framework: The node identifier (for addressing nodes using Pastry), the ensemble identifier (for creating new roles and retrieving existing roles) and the role identifier (which uniquely identifies one role instance).

Behavioral Aspects This discussion already brings us to the behavioral aspects of the framework. Two functions of the framework were already mentioned – `create` and `get`. They are implemented as the Java methods (`createRoleInstance` and `getRoleInstance`) which both perform a full network round-trip between two Pastry nodes: They require a node and an ensemble ID as well as the class of the required role as input. The target node is instructed to create and start the new role (or retrieve it, in the second case). A role identifier as discussed above is returned which can then be used for role-to-role message routing.

The behavioral specifications make heavy use of role-to-role communication. A role must be able to send a message and to expect to receive a certain message in its behavior. For this purpose the framework provides the two methods `sendMessage()` and `waitForMessage()` for communication between roles.

The method `sendMessage()` takes a message and a target role; the message is routed between Pastry nodes to an input buffer in the target role. The method only returns when this has been successfully completed (i.e., an internal acknowledge is sent back upon which the `sendMessage()` function returns normally). Otherwise, an exception is raised. Of course, safe communication requires that any message is finally consumed from the buffer. Moreover, from the semantic point of view it is required that any consumed message is also expected

by the target role as an input message in accordance with its role behavior specification. For this purpose we perform behavioral compatibility checks between role behaviors already during the ensemble modeling phase as discussed in Sec. 4.

The second method is `waitForMessage()` which instructs the framework to wait for a message of a certain type, or a selection of certain different types. The latter is required, for example, in the `Initiator` role when waiting for one of three possible messages in state i4 in Fig. 3a. The `waitForMessage()` function also takes a timeout value; an exception is raised if a message does not arrive in the given time (though specifying `INFINITY` is an option).

Given the basic infrastructure for role management and the communication functions above, we can now proceed to the role implementations.

5.2 Implementing Roles

As discussed above, role (types) are implemented in Java using classes. Thus, for each of the six roles above, a class is created, inheriting from an abstract role template for easier access to framework methods. Each role is instantiated within a certain ensemble and node. Upon startup, the main method implementing the role behavior is called.

The role behavior specifications are translated to message exchanges. For each message type, a message class with an appropriate name is created, and equipped with the required parameters as indicated in the role types. For example, the `execute` message shared between `PotentialExecutor` and `Executor` is implemented by an instance of the `ExecuteMessage` class which carries the application name as a field.

A role behavior is translated into Java as follows:

- Transitions with incoming messages, e.g. `?store()(app)`, are translated into a `waitForMessage()` framework call for the corresponding message class, e.g. `StoreApplicationMessage`. The `waitForMessage()` method returns an instance of the message once received, which can be queried for the actual app.
- Transitions with an outgoing message, e.g. `!init()(name,reqs)`, are translated into a `sendMessage()` framework call. The message to be sent must be given as a parameter.
- Transitions referring to the two framework functions `get` and `create` are directly translated to calls to the corresponding framework methods `getRoleInstance()` and `createRoleInstance()`. They return role IDs which can then be used for communication.
- All other transitions, as well as loops and decisions are translated into their appropriate Java counterparts.

With this basic description, most of the role behaviors are directly translatable into Java code. Thus, many communication errors are avoidable by a careful analysis of the ensemble model. Nevertheless, we were interested in a robust system implementation and hence we followed a defensive strategy such that all possible types of errors are taken into account.

14

One issue in the implementation is that each of the framework methods may fail for various reasons, and the resulting exceptions must be handled. Firstly, in all operations, timeouts may occur if a message could not be delivered. Secondly, role-to-role messages may fail if the target node does not (yet) participate in the expected ensemble or does not (yet) play an expected role; this also applies to the getRoleInstance() method. The createRoleInstance() may fail if the role class could not be instantiated or started. These errors are not captured in the role behaviors, but may occur in practice (in particular, they may occur during development if the implementation is not yet fully complete and stable).

A second issue is bootstrapping, both of HELENA ensembles and of basic node identification. At each ensemble startup, at least one role needs to be instantiated by an outside party before messages can be received. In this case study, the main entry point is the Deployer role; a second entry point is the Requester role. The bootstrapping point cannot be deduced from the local behavior specifications and therefore must be treated individually outside of the framework. In the case of the SCP, this part is played by the SCP UI (top right in Fig. 1).

There are also some points where the roles need to return information to an outside party. For example, the Requester role is invoked each time a UI request is made for an app; the response from the application must be presented to the user. This is exactly the opposite of the bootstrapping problem and requires explicit invocation of an outside party from the role. One could think of specialized actions for this; or introduce answers a role in general gives to users.

Basic node identification is another topic of interest: To create a role, the ID of the target node must be known. In the case of the SCP, we heavily rely on the fact that the Initiator and Storage node ID can always be found using the app name (as explained in Sec. 2). This makes both of these roles communication hubs. If such a mechanism is not available, other forms of node ID retrieval need to be found; one example is the Initiator role which uses the underlying, gossip-provided node information as an ID source. A similar problem applies to finding ensembles: A node which does not currently have a role in an ensemble does not know the ensemble ID and thus cannot route messages, which might occur in a formerly non-associated node on which a Requester is instantiated. We solve this again by using the app name as a hash for the ensemble ID, but this might be difficult in other settings.

6   Conclusion

We have shown how the HELENA modeling approach can be applied to a larger software system. Starting from the description of our case study, the Science Cloud Platform, we developed an ensemble specification based on six collaborating roles. An instance of this specification is able to deploy and execute a software application in a voluntary peer-to-peer network. Splitting the task of app execution in several independent roles was quite natural and helped to understand the individual subtasks. Compared to the development of one big component which combines all behaviors at one place, it was straightforward

15

to derive behaviors for each role individually. However, we experienced that the granularity when deciding which roles to introduce was not always clear. Using the HELENA modeling approach allowed us to examine the modeled system for communication errors before implementation. During implementation of the model, translating the role behaviors to Java code has proven to be straightforward. To gain this complexity reduction, first a (reusable) HELENA framework layer was needed to provide HELENA-specific functionalities. The encapsulation of responsibilities in separate roles helped to make the SCP code clean and easy to understand. Special care had to be taken in four areas: Handling faults during communication, node identification for role creation and retrieval, handling node failures, and communication between ensembles and the outside world.

In the future, we want to pursue different research directions. In [9], we have given a formal semantics for ensemble specifications in terms of ensemble automata. In a next step we want to define rules for the generation of an ensemble automaton from an ensemble specification based on the new process expressions for role behaviors. Secondly, based on the ensemble automaton, we want to define when an ensemble can be considered communication-safe (for static architectures, called assemblies, this has been considered in [10]). We want to investigate conditions under which communication-safety of an ensemble automaton can be derived from pairwise behavioral compatibility of role behaviors. Thirdly, we want to support the composition of large ensembles from smaller ones and to study which properties can be guaranteed for the composed system. Lastly, we want to construct an infrastructure for HELENA models that can cope with unreliable systems and failing components.

Combining the three paradigms of cloud computing, voluntary computing, and peer-to-peer computing has started to attract attention in recent years. Most approaches bridge volunteer and cloud computing for infrastructure-as-a-service systems. Cunsolo et al. [5], and Chandra and Weissman [4], they propose to use distributed voluntary resources with an architecture similar to our three-layered approach, but with a centralized management subsystem. Advocating a "fully decentralized p2p cloud", Babaoglu et al. [2] implement a system very similar to the SCP. They also introduce the idea of partitioning the system in slices matching a user's request. The idea is to create a subcloud in the system providing resources for one task. This resembles our approach of assembling nodes in task-oriented ensembles. With HELENA, we offer a modeling method for describing such task-oriented groups with a solid semantic foundation. Modeling evolving objects with roles as perspectives on the objects has been proposed by various authors [13,17], but they do not see them as autonomic entities with behavior as we do in HELENA. For describing dynamic behaviors, we share ideas with different process calculi [6,8], but we use dynamic instance creation for roles on selected components. The idea to describe structures of interacting objects without having to take the entire system into consideration was already introduced by several authors [11,3,15], but they do not tackle concurrently running ensembles of autonomic entities. We believe that these frameworks were never employed to implement a (role-based, voluntary and peer-to-peer) cloud computing platform.

References

1. The ASCENS Project, http://www.ascens-ist.eu
2. Babaoglu, Ö., Marzolla, M., Tamburini, M.: Design and implementation of a P2P Cloud system. In: Symposium on Applied Computing. pp. 412–417. ACM (2012)
3. Baldoni, M., Studi, U., Italy, T.: Interaction between Objects in powerJava. Journal of Object Technology 6, 7–12 (2007)
4. Chandra, A., Weissman, J.: Nebulas: Using Distributed Voluntary Resources to Build Clouds. In: Conf. on Hot Topics in Cloud Computing. USENIX Association (2009)
5. Cunsolo, V.D., Distefano, S., Puliafito, A., Scarpa, M.: CloudHome: Bridging the Gap between Volunteer and Cloud Computing. In: Int. Conf. on Intelligent Computing. pp. 423–432. LNCS, Springer (2009)
6. De Nicola, R., Ferrari, G.L., Loreti, M., Pugliese, R.: A Language-Based Approach to Autonomic Computing. In: Beckert, B., Damiani, F., de Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects. LNCS, vol. 7542, pp. 25–48. Springer (2011)
7. Demers, A.J., Greene, D.H., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H.E., Swinehart, D.C., Terry, D.B.: Epidemic algorithms for replicated database maintenance. In: Symposium on Principles of Distributed Computing. pp. 1–12. ACM (1987)
8. Deniélou, P.M., Yoshida, N.: Dynamic Multirole Session Types. In: Symposium on Principles of Programming Languages. pp. 435–446. ACM (2011)
9. Hennicker, R., Klarl, A.: Foundations for Ensemble Modeling - The Helena Approach. In: Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi. LNCS, Springer (to appear 2014)
10. Hennicker, R., Knapp, A., Wirsing, M.: Assembly theories for communication-safe component systems. In: From Programs to Systems - The Systems Perspective in Computing. LNCS, vol. 8415. Springer (to appear 2014)
11. Herrmann, S.: Object Teams: Improving Modularity for Crosscutting Collaborations. In: Int. Conf. NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World. pp. 248–264. Springer (2003)
12. Klarl, A., Hennicker, R.: Design and Implementation of Dynamically Evolving Ensembles with the Helena Framework. In: Australasian Software Engineering Conf. IEEE (to appear 2014)
13. Kristensen, B.B., Østerbye, K.: Roles: Conceptual Abstraction Theory and Practical Language Issues. Theor. Pract. Object Syst. 2(3), 143–160 (1996)
14. Mayer, P., Klarl, A., Hennicker, R., Puviani, M., Tiezzi, F., Pugliese, R., Keznikl, J., Bureš, T.: The Autonomic Cloud: A Vision of Voluntary, Peer-2-Peer Cloud Computing. In: Wshp. on Challenges for Achieving Self-Awareness in Autonomic Systems. pp. 1–6. IEEE (2013)
15. Reenskaug, T.: Working with objects: the OOram Framework Design Principles. Manning Publications (1996)
16. Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Int. Conf. on Distributed Systems Platforms. pp. 329–350. Springer (2001)
17. Steimann, F.: On the representation of roles in object-oriented and conceptual modelling. Data Knowl. Eng. 35(1), 83–106 (2000)
18. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 2nd edn. (2002)