Foundations for Ensemble Modeling - The HELENA Approach

Handling massively distributed systems with ELaborate ENsemble Architectures*

Rolf Hennicker and Annabelle Klarl

Ludwig-Maximilians-Universität München
Germany

Abstract. Ensembles are groups of active entities that collaborate to perform a certain task. Modeling software systems for ensemble execution is challenging since such applications are highly dynamic involving complex interaction structures of concurrently running individuals. In this work, we propose a formal foundation for ensemble modeling based on a rigorous semantic framework. Our approach is centered around the notion of a role expressing the capabilities that a component needs when participating in a specific ensemble. We use ensemble structures to model the structural aspects of collaborations and labeled transition systems to specify the dynamic behavior typical for performing a certain role. Our approach is driven by a clear discrimination between types, used on the specification level, and instances, which form concrete ensembles in an ensemble automaton. The semantics of an ensemble specification is given by the class of all ensemble automata which adhere to the properties of an ensemble structure such that any ensemble member, playing a certain role, exhibits a behavior that is allowed by the role behavior specification.

1 Introduction

1.1 Motivation

The continuously increasing potential of new computer technologies paves the way for developing advanced applications in which huge numbers of distributed nodes collaborate to accomplish various tasks under changing environments. Application domains are, for instance, environmental monitoring and simulation, robotics, e-mobility and cloud computing. Such applications are typically highly dynamic involving a complex interaction behavior between nodes. Nodes may join or leave a collaboration, they may change location and they may autonomously adapt to new conditions. Systems supporting such applications are extremely software-intensive. In contrast to available hardware, current software engineering practices are not sufficiently developed to support such scenarios in

* This work has been partially sponsored by the EU project ASCENS, 257414.

a reliable way on a semantically solid basis with sound formal specification and verification techniques.

On this background, the EU project ASCENS [1,34] pursues the goal to develop foundations, techniques and tools to support the whole life cycle for the construction of Autonomic Service-Component ENSembles [9]. An ensemble is understood as a collection of autonomic entities that collaborate for some global goal. Following [27,2], a goal can be an “achieve goal”, such that the ensemble will terminate when the goal (specified, e.g., by a particular state) is reached, or a “maintenance goal”, such that a certain property (specified, e.g., by a system invariant) is maintained while the system is running.

The inherent complexity and dynamics of ensembles exhibiting a collective, goal-oriented behavior is a huge challenge. Well-known techniques, like component-based software engineering [33,31], are not sufficient for modeling ensembles, but must be augmented with other features that allow to focus on the particular characteristics of ensembles. While a component model describes the architectural and dynamic properties of a (complex) target system, ensembles are dynamically formed on demand as specific, goal-oriented communication groups running on top of a target system and different ensembles may run concurrently on the same system (dealing with different tasks). The target platform of the system can be component-based, but it is crucial to recognize that the same component instance may take part in different ensembles under particular, ensemble-specific *roles*. A component instance can play different roles at the same time and it can dynamically change its role. Therefore, we propose to center our approach around the notion of a role [21] and to model an ensemble in terms of roles and their interactions to collectively pursue a certain goal.

Ensemble modeling is particularly important in the analysis phase of the development life cycle since it allows us to concentrate only on parts of the capabilities that a component must finally support. Each role a component can fill represents a particular view on the component needed to solve a specific collaborative task. In this way complexity of system modeling can be significantly reduced.

1.2 The Helena Approach

In this paper, we propose a rigorous formal foundation for ensemble modeling that can be used during requirements elicitation and as a basis for the development of designs. In the Helena approach, we assume given a set of component types. The component types define basic attributes and operations that are commonly available. Each role (more precisely, role type) is defined for a subset of component types whose instances can fill the role. A role specifies particular capabilities in terms of role attributes and role operations that are only relevant when performing the role. The structural aspects of a collaboration are determined by an *ensemble structure* which consists of a set of roles (constrained by multiplicities) and a set of role connectors determining which roles may interact in terms of which operations. This introduces a level of security since other interactions are not legal; i.e. an interaction requested by a component which does

not fit to its current role would be a failure. For visualizing ensemble structures, we use UML-like notations [30]. Additionally, we use labeled transition systems to determine the dynamic aspects of a collaboration in terms of role behaviors such that collaboration is directed towards a specific task.

Our framework supports specialization in the sense that different extensions and interpretations of an ensemble model are possible. For instance we do not fix any particular paradigm for interaction on the level of an ensemble structure. Interaction could be performed by accessing knowledge in the repositories of components, like in SCEL [17,18], it could be realized by implicit knowledge exchange managed by the runtime infrastructure, like in DEECo [12], or it could be based on explicit synchronous or asynchronous communication.

To provide semantics for an ensemble specification, the interaction paradigm must be instantiated. In this paper, we show how this can be done for the case of synchronous message passing systems. For a given ensemble specification, we consider the class of its semantic models given by particular labeled transition systems called *ensemble automata*. Each state of the system determines a set of component instances which are currently participating in the ensemble and a set of role instances which are currently adopted by the component instances. Both component and role instances have a current data state determined by their attribute values respectively. The attribute values of a component instance ci determine the (basic) information that is shared by all role instances that ci is currently playing. Moreover, to each role instance a control state is associated that determines its current progress according to the behavior specification of the corresponding role type. Transitions between ensemble states are caused either by communication between role instances according to a role connector or when certain management operations are performed such that component instances join or leave an ensemble, change their role or adopt an additional role.

In the following sections, we first consider, in Sect. 2, the syntactic notions for ensemble structures and ensemble specifications. In Sect. 3, we define their semantic interpretations: we consider ensemble states, formed by collections of component and role instances, and we focus on the particular case of synchronous message passing systems for which we introduce ensemble automata as semantic models of ensemble specifications. In Sect. 4, we discuss related work and, in Sect. 5, we give a short summary and point out ideas how our approach will be extended towards a comprehensive, semantically well-founded ensemble development methodology.

Dedication. Our approach is strongly influenced by the school of algebraic specifications and institutions [20], including the seminal work of Prof. Futatsugi as one of the leading architects of prominent algebraic specification languages like OBJ2 [19] and CafeOBJ [29]. Indeed, an ensemble structure in HELENA can be considered as a signature, ensemble automata as models of that signature, ensemble states as (higher-order) algebras, ensemble specifications as presentations and the satisfaction relation is implicitly given by the notion of a model of an ensemble specification. We would like to thank Prof. Futatsugi very cordially for his important contributions to the field and for his very friendly attitude in

scientific and private discussions. In particular, it was always a great pleasure to discuss with him new ideas for the observational interpretation of algebraic specifications, as supported by CafeOBJ and implemented in the CafeOBJ environment [28]. It is a pleasure for us to dedicate this work to Prof. Futatsugi and we want to wish him many more new exciting ideas and experiences in the future.

2 Ensemble Structures and Specifications

In the HELENA approach, we tackle systems with a large number of entities which collaborate towards a specific goal. The foundation for those systems are components which are presented in the first subsection. To cope with the complexity of systems with large numbers of components, we afterwards introduce the notion of an ensemble structure as a view on a component-based system. Lastly, we outline the specification of the dynamic behavior of roles collaborating in such an ensemble structure to direct behavior towards the intended task.

Throughout the paper, we use a peer-2-peer network as running example which supports the distributed storage of files that can be retrieved upon request. Several peers of the network will work together when a file is requested. One peer will play the role of the requester of the file, other peers will act as routers and finally, the peer storing the requested file will appear in the role of a provider.

Notation. Whenever we consider tuples $t = (t_1, \dots, t_n)$, in the following we use the notation $t_i(t)$ to refer to t_i.

2.1 Components

First, we introduce the concepts of rudimentary components providing basic information usable in all roles the component can fill. Component types are characterized by *attributes* and *operations*. Attributes and parameters of operations are not (necessarily) typed.

Definition 1 (Attributes and Operations). *An* attribute *is a named variable. An* operation *op is of the form $op = opname(params)$ such that $opname$ is the name of the operation and $params$ is a list of formal parameters.*

Definition 2 (Attribute Values). *Let A be a set of attributes and $\mathcal{D}$ a universe of data values. An* A-state *is a function $\delta : A \to \mathcal{D}$ which assigns a value in $\mathcal{D}$ to each attribute in A. The set of all A-states is denoted by $DStates_A$.*

Let us consider this definition in the context of our running example of a peer-2-peer network. Typical attributes in such an environment are the network address of an entity and the list of filenames and their content which an entity stores. The set of attributes can thus be defined as $A = \{$`address`, `fileNames`, `contents`$\}$. The function δ_1 may, for example, assign the value `198.121.1.3` to the attribute `address`, `[1.txt, 2.pdf]` to the attribute `fileNames`, and some file contents to the attribute `contents`.

To classify components according to their capabilities, we introduce *component types*. A component type defines the attributes and operations for all components of that type. It forms the basis for more specialized and complex capabilities. A component instance is a concrete instantiation of its component type.

Definition 3 (Component Type). *A* component type *ct is a tuple $ct = (nm, attrs, ops)$ such that nm is the name of the component type, $attrs$ is a set of attributes, and $ops = \langle ops_{out}, ops_{in}, ops_{int} \rangle$ with ops_{out}, ops_{in}, and ops_{int} are sets of outgoing, incoming, and internal operations respectively.*

The basic component type in a peer-2-peer network is $peer = (\texttt{Peer}, \{\texttt{address}, \texttt{fileNames}, \texttt{contents}\}, \langle \emptyset, \emptyset, \emptyset \rangle)$. Each component of component type *peer* has the attributes `address`, `fileNames`, and `contents` and no (basic) operations, since all *peer* operations introduced in the sequel will only be relevant for particular roles. For visualization, we introduce a graphical notation for component types like in UML (cf. Fig. 1).

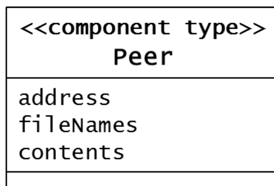


Fig. 1: Component type *peer* in graphical notation

2.2 Ensemble Structures

Components can collaborate to perform certain tasks. For this purpose, they 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. A role (more precisely, role type) defines which types of components can contribute the desired functionality to the overall collaboration and enhances them with role-specific capabilities. Firstly, the role specifies the component types of entities which are able to fill this role. Secondly, it defines role-specific attributes to store data that is relevant for performing the role and role-specific operations which are required to fulfill the responsibilities of the role.

Definition 4 (Role). *Let CT be a set of component types. A* role r over CT *is a tuple $r = (head, roleattrs, roleops)$ such that*

- *$head = \langle nm, ctypes \rangle$ declares the name nm of the role together with a finite, non-empty set $ctypes \subseteq CT$ of component types (whose instances can fill the role r),*
- *$roleattrs$ specifies the role specific attributes, and*
- *$roleops = \langle roleops_{out}, roleops_{in}, roleops_{int} \rangle$ specifies outgoing, incoming, and internal operations provided by the role r.*

In the context of our peer-2-peer network, we consider the task of requesting and transferring a file. To perform this task, we envision three roles: requester, router, and provider. The requester wants to download the file. First, it needs to request the address of the peer storing the file from the network, while using the routers as forwarding peers of its request. Once the requester knows the address, it directly requests the file from the provider for download. Each role can be adopted by instances of component type *peer*, but exhibits different capabilities to take over responsibility for the transfer task. The requester must be able to request the address of the provider from a router and receive the reply. Afterwards, it must be able to request the file from the provider and receive the content. The router must be able to receive a request for the address, forward it to another router, receive the reply from another router, and send it back. The provider of a certain file must be able to receive a request for the file and send back the content. Formally, the role of the provider peer is defined as follows:

$$\begin{aligned} provider = (\langle\ \texttt{Provider}, \{peer\}\ \rangle, \emptyset, \\ \langle \{\texttt{sndFile(cont)}\}, \{\texttt{reqFile(fn)}\}, \emptyset \rangle) \end{aligned}$$

Note that for this role neither specific attributes nor internal operations are necessary, but the requester role stores the name of the requested file in its role-specific attribute `fileName`.

We use a UML-like visualization of roles annotated with the stereotype `«role type»`. The diagrams for the roles in the peer-2-peer network are given in Fig. 2. They consist of three parts: the name of the role followed by the set of component types which can fill the role, the role attributes, and the role operations together with the modifiers `out`, `in`, and `int`.

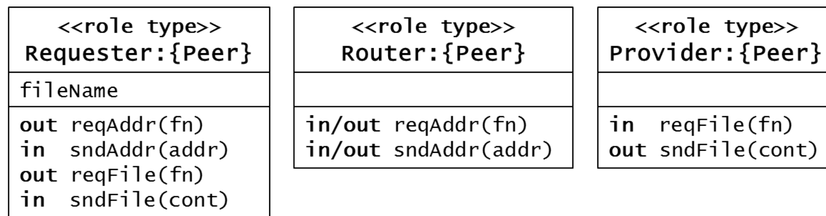


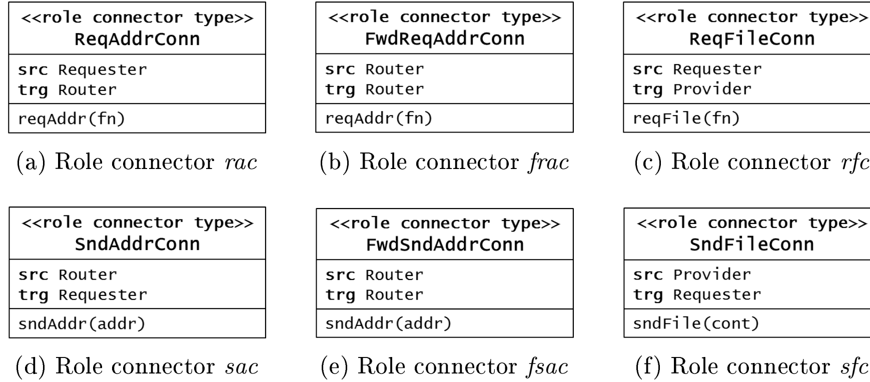
Fig. 2: Roles *requester*, *router*, and *provider* in graphical notation

To collaborate on tasks, roles need to communicate. A role initiates the information transfer via the call of an outgoing operation and receives information via the reception of an incoming operation. However, for the specification of collaborations we do not only want to declare communication abilities of a single role, but also to specify which roles are meant to interact by which messages. This information is specified by a *role connector* (or more precisely, role connector type). Role connectors are directed such that they can also support multicast sending of messages.

Definition 5 (Role Connector). *Let CT be a set of component types and R be a set of roles over CT. A* role connector *rc* over R *is a tuple* $rc = (nm, src, trg, ops, rcconstraints)$ *such that*

- nm *is the name of the role connector,*
- $src \in R$ *denotes the source role from which information is transferred along rc,*
- $trg \in R$ *denotes the target role to which information is transferred along rc, and*
- ops *is a set of operations such that* $ops \subseteq roleops_{out}(src) \cap roleops_{in}(trg)$ *determine which messages can be sent along rc.*

In our running example, a requester peer needs to send a download request for a file to the provider peer. For that communication, we introduce the role connector $rfc = ($`ReqFileConn`$, requester, provider, \{$`reqFile(fn)`$\})$. In Fig. 2 we can verify that *rfc* is well-formed according to Def. 5 since `reqFile` is an outgoing operation for the role *requester* and an incoming operation for the role *provider*. For the reply, we introduce the role connector $sfc = ($`SndFileConn`$, provider, requester, \{$`sndFile(cont)`$\})$. Role connectors are visualized as shown in Fig. 3. The first box shows the name of the role connector, the second one the source and target role, and the last one the exchanged messages. Although in our example *rfc* and *sfc* are only responsible for one message, role connectors can in general allow a set of messages, some of which could also be declared as multicast messages.



(a) Role connector *rac* (b) Role connector *frac* (c) Role connector *rfc*

(d) Role connector *sac* (e) Role connector *fsac* (f) Role connector *sfc*

Fig. 3: Role connectors providing interaction abilities in graphical notation

Roles and role connectors form the basic building blocks for collaborations in ensembles. An *ensemble structure* determines the kind of teams needed to perform a task. An ensemble structure specifies which roles contribute to the collaboration and which role connectors are required for interaction. Additionally, in an ensemble structure roles are equipped with a multiplicity which determines how many instances may contribute. Thus, an ensemble structure specifies the structural aspects of a collaboration.

Definition 6 (Ensemble Structure). *Let CT be a set of component types. An* ensemble structure Σ over CT *is a pair* $\Sigma = (roles, conns)$ *such that*

- *roles is a set of roles over CT such that each* $r \in roles$ *has a multiplicity* $mult(r) \in Mult$ *and Mult is the set of multiplicities available in UML, like* $0..1$ *or* $*$,
- *conns is a set of role connectors over roles such that for each* $rc \in conns$*, it holds* $src(rc), trg(rc) \in roles$.

The ensemble structure Σ *is* closed *if all operations are used in connectors, i.e. if*

$$\bigcup_{rc \in conns} ops(rc) = \bigcup_{r \in roles} (roleops_{out}(r) \cup roleops_{in}(r));$$

otherwise it is open.

For our peer example, we define an ensemble structure $\Sigma_{transfer}$. The ensemble structure is composed of a requester role (with at most one instance participating in the ensemble), a router role (with arbitrarily many instances participating in the ensemble), and a provider role (with at most one instance participating in the ensemble). Communication between those roles is needed to request and receive the provider address from the network (possibly involving several forwarding steps via routers) and finally to request and receive the file from the provider itself. Formally, the ensemble structure $\Sigma_{transfer} = (roles, conns)$ embraces the two sets:

$$\begin{aligned} roles &= \{\langle requester, 0..1\rangle, \langle router, *\rangle, \langle provider, 0..1\rangle\} \\ conns &= \{rac, sac, frac, fsac, sfc, rfc\} \end{aligned}$$

In the set *roles*, we find each role associated with a multiplicity as mentioned before. The role connectors in the set *conns* provide the means to request and send address and file (cf. Fig. 3). We visualize ensemble structures similarly to collaborations in composite structure diagrams in UML 2. Fig. 4 shows the ensemble structure $\Sigma_{transfer}$ in graphical notation. Roles are depicted as boxes with the multiplicity written in the upper right corner. Role connectors are represented as arrows between source and target roles labeled with the connector name.

2.3 Ensemble Specifications

After having modeled the structural aspects of an ensemble, we move on to the specification of dynamic behaviors. A role itself declares the particular capabilities needed to perform a certain task in the form of its operations. How these operations are used to model role behavior is formalized by a *labeled transition system*. Starting from an initial state, the role behavior specifies which sequences of operations can be executed to contribute the required responsibilities of this role to the overall collaboration.

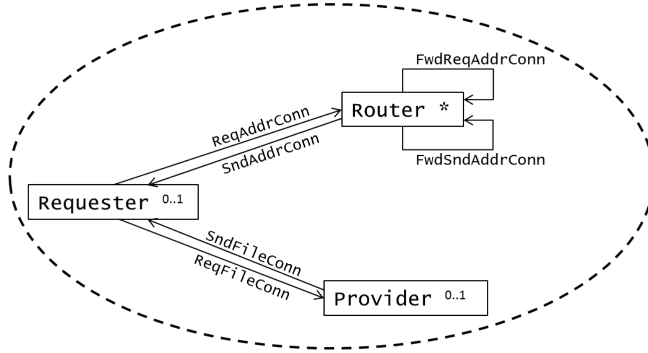


Fig. 4: Ensemble structure $\Sigma_{transfer}$

Definition 7 (Labeled Transition System). *A* labeled transition system (LTS) *is a tuple* $(Q, q_0, \Lambda, \Delta)$ *such that* Q *is a set of states,* $q_0 \in Q$ *is the initial state of the LTS,* Λ *is a set of labels, and* $\Delta \subseteq Q \times \Lambda \times Q$ *is a transition relation. For* $(q, l, q') \in \Delta$*, we also write* $(q \xrightarrow{l} q') \in \Delta$*.*

A *role behavior* is a labeled transition system whose labels denote sending an operation (expressed by the operation followed by an exclamation mark "!") or receiving an operation (expressed by the operation followed by a question mark "?") or executing an internal operation (expressed just by the operation).

Definition 8 (Role Behavior). *Let* $\Sigma = (roles, conns)$ *be an ensemble structure and* $r \in roles$*. A* role behavior *of* r *is given by a labeled transition system* $RoleBeh_r = (Q, q_0, \Lambda, \Delta)$ *such that*

- Q *is a set of control states,*
- $q_0 \in Q$ *is the initial state,*
- Λ *is the set of labels given by*
  $\{nm(rc).op! \mid \exists rc \in conns : r = src(rc),\ op \in ops(rc)\} \cup$
  $\{nm(rc).op? \mid \exists rc \in conns : r = trg(rc),\ op \in ops(rc)\}$,
- $\Delta \subseteq Q \times \Lambda \times Q$ *is a transition relation.*

Following our notational convention, we write $Q(RoleBeh_r)$ for Q, $q_0(RoleBeh_r)$ for q_0, $\Lambda(RoleBeh_r)$ for Λ, and $\Delta(RoleBeh_r)$ for Δ.

The full specification of an ensemble comprises the architecture of the collaboration in terms of an ensemble structure Σ and the set of all role behavior specifications.

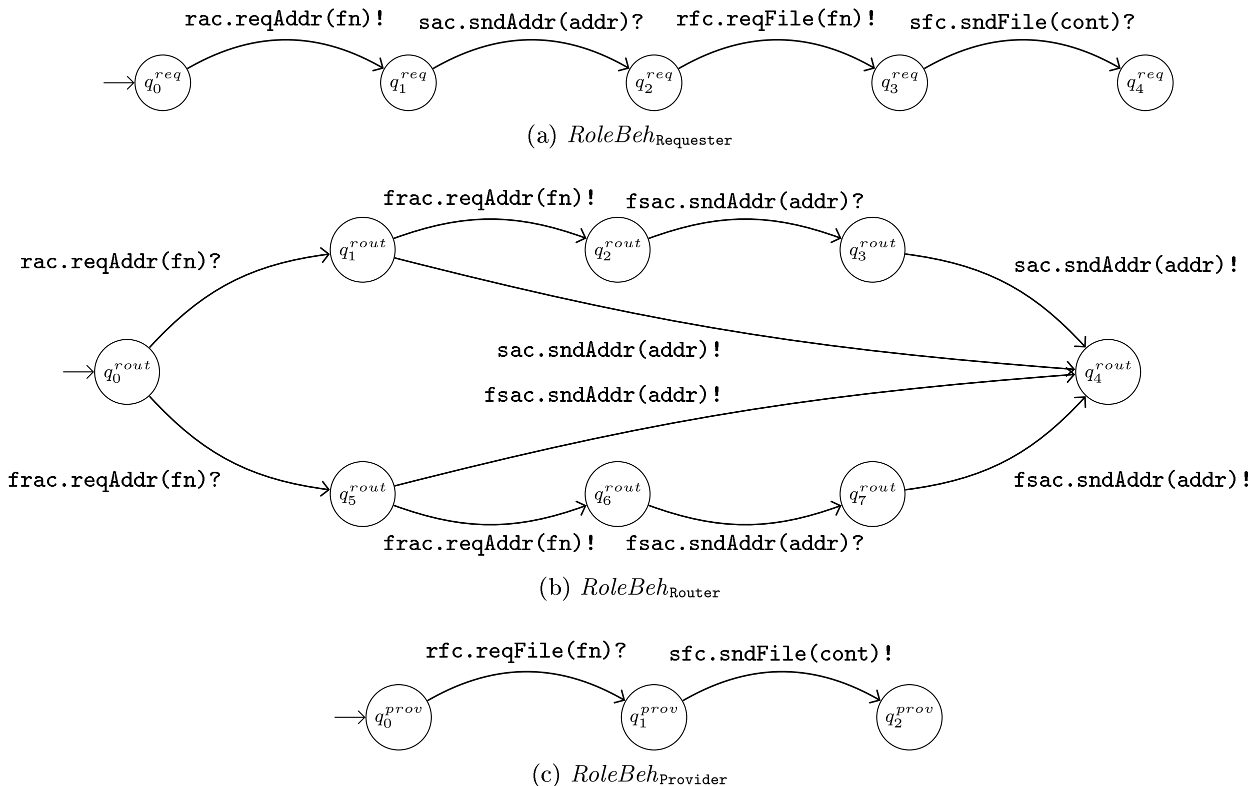
Definition 9 (Ensemble specification). *An* ensemble specification *is a pair* $EnsSpec = (\Sigma, RoleBeh)$ *such that*

- $\Sigma = (roles, conns)$ *is an ensemble structure over a set* CT *of component types, and*
- $RoleBeh = (RoleBeh_r)_{r \in roles}$ *is a family of role behaviors* $RoleBeh_r$ *for each* $r \in roles$.

Let us illustrate the specification of ensembles with our running example. We specify the dynamic behavior of the *requester*, *router* and *provider* roles by the three role behaviors $RoleBeh_{requester}$, $RoleBeh_{router}$, and $RoleBeh_{provider}$ shown in Fig. 5. All three behaviors terminate since in this application we consider an achieve goal such that an ensemble stops when it has fulfilled its task.

The router role exhibits the most interesting behavior. Its responsibility is to provide the address of the provider to a requesting peer. A router can first receive a request `reqAddr(fn)?` to search for the address where the file with name `fn` is located from a requester, using the connector rac, or from (another) router, using the forward request address connector $frac$. Since the router may or may not store the file itself, in each case it has two possibilities to proceed: either it has the file and thus sends its own address back to the requester with the message `sndAddr(addr)!`, or it does not have the file and thus requests the address from a neighboring peer by issuing the call `reqAddr(fn)!`. In the first case, it has immediately met its responsibility according to the router role while in the second case it has to wait for a response and then to forward it to the requesting peer. Note that on the instance level considered later on in Sect. 3, the peer instance playing the router will adopt the role of a provider when it detects that it stores the file itself (cf. the transition from state σ_5 to σ_6 in Fig. 9).

With this example we want to illustrate that the separate consideration of roles facilitates significantly the task of system specification for ensembles. If we had directly started with component-based modeling of a peer component, it would have been necessary to specify the full component behavior at once. This behavior would have to model all possible behaviors which a component instance should be able to perform. In particular, one would have to decide whether a component instance should administrate several threads for concurrent executions of different tasks at the same time or whether a component is only able to perform different tasks in a sequential order.



(a) $RoleBeh_{\texttt{Requester}}$

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

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

Fig. 5: Behaviors specifications

3 Semantics of Ensemble Specifications

3.1 Ensemble States

An ensemble structure $\Sigma = (roles, conns)$ over a set CT of component types specifies which roles and role connectors are needed to perform a task. The actual execution of an ensemble will be performed by collaborating component instances. We assume that before an ensemble is started there exists already a repository of component instances which can potentially contribute. Formally, this repository is given by a family $INST = (INST_{ct})_{ct \in CT}$ of pairwise disjoint sets $INST_{ct}$ of component instances for each component type ct. For an ensemble state σ, the currently participating component instances are determined by the sets $insts = (insts_{ct})_{ct \in CT}$ shown in Def. 10 below. The situation is different for roles. Role instances are only created when a component instance adopts that role. Formally, we assume given a family $RID = (RID_r)_{r \in roles}$ of countably infinite and pairwise disjoint sets RID_r of role identifiers for role r. These sets determine a space of names which can be instantiated when a new role instance is created. An ensemble state σ has not only to record which are the current component members of the ensemble, but also which role instances currently exist. These are determined by the sets $roleinsts = (roleinsts_r)_{r \in roles}$ below. Any existing role instance must be adopted by exactly one component instance and any participating component instance must at least adopt one role instance. For the formalization of these relationships we use the surjective mappings $adoptedBy = (adoptedBy_r)_{r \in roles}$ whose functionalities are defined below.

To fully specify a Σ-ensemble state, we additionally need to determine the data states of component and role instances (given by valuations of component and role attributes resp.), and also the control state of a role instance showing the current progress of its execution. For this purpose, we use the families of functions *data*, *roledata*, and *control* as indicated below.

Definition 10 (Σ-ensemble state). *Let CT be a set of component types, $INST$ be a family of sets of component instances and RID be a universe of role identifiers as explained above. A* Σ-ensemble state *(over $INST$) is a tuple*

$$\sigma = (insts, roleinsts, adoptedBy, data, roledata, control)$$

such that

- $insts = (insts_{ct})_{ct \in CT}$ *is a family of sets* $insts_{ct} \subseteq INST_{ct}$ *of component instances currently participating in the ensemble,*
- $roleinsts = (roleinsts_r)_{r \in roles}$ *is a family of sets* $roleinsts_r \subseteq RID_r$ *of role instances currently existing in the ensemble such that the multiplicities of* $r \in roles$ *in* Σ *are respected, i.e.* $|roleinsts_r| \leq 1$ *if* $\mathrm{mult}(r) = 0..1$,
- $adoptedBy = (adoptedBy_r)_{r \in roles}$ *is a family of surjective functions* $adoptedBy_r : roleinsts_r \to \bigcup_{ct \in ctypes(r)} insts_{ct}$ *such that each role instance is associated to a unique component instance,*

- *$data = (data_{ct})_{ct \in CT}$ is a family of functions*
  $data_{ct} : insts_{ct} \to DStates_{attrs(ct)}$,
- *$roledata = (roledata_r)_{r \in roles}$ is a family of functions*
  $roledata_r : roleinsts_r \to DStates_{roleattrs(r)}$,
- *$control = (control_r)_{r \in roles}$ is a family of functions*
  $control_r : roleinsts_r \to CStates_r$ with a set $CStates_r$ of control states.

The set of all Σ-ensemble states is denotated by $States_\Sigma$.

Following our notational conventions, for a Σ-ensemble state $\sigma = (insts, roleinsts, adoptedBy, data, roledata, control)$ we write $insts(\sigma)$ for $insts$, $insts_{ct}(\sigma)$ for $insts_{ct}$, and similarly for all other parts of σ.

To illustrate the meaning of the *adoptedBy* functions, we visualize the mapping for two different Σ-ensemble states σ_1 and σ_2 in Fig. 6. Both states are based on the set $INST = \{$`ci1`, `ci1'`, `ci2'`$\}$ of component instances of type `CT` and `CT'` resp.. The idea is that there are two ensembles running in parallel such that σ_1 is a state of the first ensemble and σ_2 is a state of the second. The given component instances should be able to participate at the same time in both ensembles. For instance in σ_1, `ci1` adopts the role instances `ri1` and `ri1'` of different role types `R` and `R'`. In σ_2, `ci1` adopts, at the time, another role instance `ri3` of type `R`. Being surjective, each function $adoptedBy_r$ associates each component instance *ci* which is participating in an ensemble with at least one role instance. The inverse image of one component instance *ci* in a particular state is thus the set of all role instances which *ci* is currently playing in that state. Only component instances that do currently not participate in an ensemble, like `ci1'` in Fig. 6, have no associated role instance.

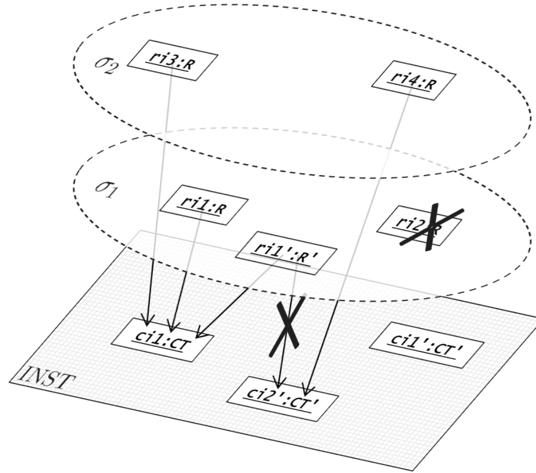


Fig. 6: Visualization of the function *adoptedBy*

Let us illustrate the definition of a Σ-ensemble state at our peer-2-peer network. Consider the ensemble structure $\Sigma_{transfer}$ and four component instances of type *peer* such that $INST = INST_{peer} = \{p1, p2, p3, p4\}$, i.e. we have given a system with four peers. A valid ensemble state over $INST$ could be that $p1$ has adopted the role of a requester that requests a file with name `"song.mp3"`, $p2$ and $p3$ work as routers, and $p3$ provides the file; $p4$ is not involved in this collaboration. The formal representation of such a $\Sigma_{transfer}$-ensemble state $\sigma = (insts, roleinsts, adoptedBy, data, roledata, control)$ is given in Fig. 7; a graphical representation of this state is shown in Fig. 8. The current control state of each role instance is shown in a circle and taken from the role behavior specifications. For instance, $rout1$ being in control state q_2^{rout} has just sent out a request address message to another router via the role connector $frac$, and $rout2$ being in control state q_5^{rout} has just received this message. We assume that the component $p3$ stores the requested file and therefore adopts, in the current state, also the role of a provider being in the initial provider state q_0^{prov}.

$$
\begin{array}{ll}
insts_{peer} & = \{p1, p2, p3\} \\
roleinsts_{requester} & = \{req\} \\
roleinsts_{router} & = \{rout1, rout2\} \\
roleinsts_{provider} & = \{prov\} \\
adoptedBy_{requester}(req) & = \{p1\} \\
adoptedBy_{router}(rout1) & = \{p2\} \\
adoptedBy_{router}(rout2) & = \{p3\} \\
adoptedBy_{provider}(prov) & = \{p3\}
\end{array}
\qquad
\begin{array}{ll}
data_{peer}(p1) & = \{(\texttt{address} \mapsto \texttt{198.121.1.1}, \\
 & \quad \texttt{fileNames} \mapsto \ldots)\} \\
 & \quad \texttt{contents} \mapsto \ldots)\} \\
data_{peer}(p2) & = \ldots \\
data_{peer}(p3) & = \ldots \\
roledata_{requester}(req) & = \{\texttt{fileName} \mapsto \texttt{"song.mp3"}\} \\
roledata_(_) & = \emptyset \\
control_{requester}(req) & = q_1^{req} \\
control_{router}(rout1) & = q_2^{rout} \\
control_{router}(rout2) & = q_5^{rout} \\
control_{provider}(prov) & = q_0^{prov}
\end{array}
$$

Fig. 7: A $\Sigma_{transfer}$-ensemble state σ

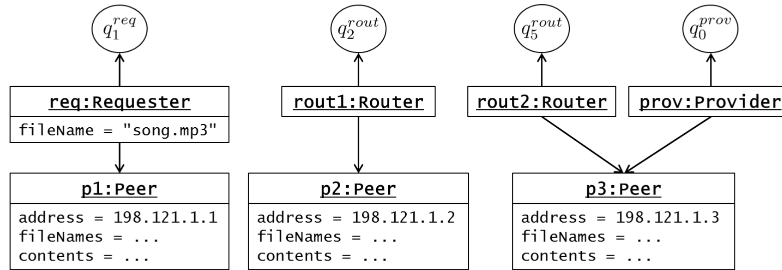


Fig. 8: $\Sigma_{transfer}$-ensemble state σ in graphical notation

3.2 Ensemble Automata

A system evolves over time by execution of operations of component instances. To model the precise collaborative behavior we have to fix how interaction is performed. In this paper, we consider the case of message passing systems with synchronous communication. Two communication partners must synchronize whenever they want to execute a shared input/output operation; otherwise they cannot proceed. We define a formal execution model for ensembles in terms of ensemble automata. Their states are ensemble states as defined in the last section. We consider two kinds of actions that can cause state transitions. First, we consider communication actions which express synchronous communication betweeen role instances. These actions are represented by operation labels of the form $opname(actparams)(rc, ri, ri')$ meaning that a role instance ri sends a message determined by an operation with name *opname* and with actual parameters *actparams* via a role connector rc to a role instance ri'. Of course, the message must be supported by the role connector and the role types of the communicating role instances must fit to the source and target roles of the connector. For technical simplicity, we assume that ensemble structures are closed and that roles and component types do not declare internal operations. The general case could be modeled by simple variants of the form of operation labels. The second kind of actions are represented by management labels of the form $adopt(ci, r)$ or $giveUp(ci, ri)$. The first label expresses that a component instance ci adopts a role r, either because ci is joining the ensemble or because ci adopts an additional role. In any case, a new role instance will be created (cf. Eq. (1)), and the *adoptedBy* function will be updated accordingly (cf. Eq. (2)). The second management label expresses that a component instance ci gives up a role instance ri. The role instance is then deleted from the ensemble and the component instance must leave the ensemble if this was the only role played by the component. For all kinds of labels, appropriate pre- and postconditions are provided that are respected by the transitions. The postconditions specify the effect of the operation for the different constituent parts of an ensemble state. If no effect is specified then the interpretation is loose leaving room for non-deterministic behavior.

Definition 11 (Σ-ensemble automaton). *Let CT be a set of component types and let $INST = (INST_{ct})_{ct \in CT}$ be a family component instances as in Def. 10. Let $\Sigma = (roles, conns)$ be an ensemble structure over CT. A Σ-*ensemble automaton (over INST) *is a labeled transition system $M = (S, \sigma_0, L, T)$ such that*

- $S \subseteq States_\Sigma$,
- $\sigma_0 \in S$ *is the initial state,*
- $L = oplabels \cup mgmtlabels$ *such that*
  - $oplabels = \{opname(actparams)(rc, ri, ri') \mid$ $rc \in conns, ri \in RID_{src(rc)}, ri' \in RID_{trg(rc)},$ $opname(params) \in ops(rc)$ *such that* $actparams \in \mathcal{D}^*$ *is a list of actual parameters instantiating* $params\}$

- $mgmtlabels =$
  $\{adopt(ci, r) \mid ci \in INST_{ct}, r \in roles \text{ such that } ct \in ctypes(r)\} \cup$
  $\{giveUp(ci, ri) \mid ci \in INST_{ct}, ri \in RID_r$
  $\text{such that } r \in roles \text{ and } ct \in ctypes(r)\}$

– *for each* $(\sigma_1, l, \sigma_2) \in T$, *one of the following holds:*

- *if* $l = opname(actparams)(rc, ri, ri')$ *then*

$$\begin{array}{ll} \textit{(pre)} & ri \in roleinsts_{src(rc)}(\sigma_1), ri' \in roleinsts_{trg(rc)}(\sigma_1), \\ \textit{(post)} & insts(\sigma_2) = insts(\sigma_1), roleinsts(\sigma_2) = roleinsts(\sigma_1), \\ & adoptedBy(\sigma_2) = adoptedBy(\sigma_1) \end{array}$$

- *if* $l = adopt(ci, r)$ *with* $ci \in INST_{ct}, r \in roles$ *then*

$$\begin{array}{lll} \textit{(post)} & insts_{ct}(\sigma_2) = insts_{ct}(\sigma_1) \cup \{ci\}, & \\ & insts_{ct'}(\sigma_2) = insts_{ct'}(\sigma_1) \text{ for all } ct' \neq ct, & \\ & roleinsts_r(\sigma_2) = & (1) \\ & \quad roleinsts_r(\sigma_1) \cup \{ri\} \text{ with } ri \in RID_r, ri \notin roleinsts_r(\sigma_1), & \\ & roleinsts_{r'}(\sigma_2) = roleinsts_{r'}(\sigma_1) \text{ for all } r' \neq r, & \\ & adoptedBy_r(\sigma_2)(ri) = ci \text{ for the new role instance } ri, & (2) \\ & adoptedBy_{r'}(\sigma_2)(ri') = & \\ & \quad adoptedBy_{r'}(\sigma_1)(ri') \text{ for all } r' \in roles, ri' \neq ri, & \\ & data(\sigma_2) = data(\sigma_1), & \\ & roledata_{r'}(\sigma_2)(ri') = & \\ & \quad roledata_{r'}(\sigma_1)(ri') \text{ for all } r' \in roles, ri' \neq ri, & \\ & control_{r'}(\sigma_2)(ri') = & \\ & \quad control_{r'}(\sigma_1)(ri') \text{ for all } r' \in roles, ri' \neq ri, & \end{array}$$

- *if* $l = giveUp(ci, ri)$ *with* $ci \in INST_{ct}, ri \in RID_r$ *then*

$$\begin{array}{ll} \textit{(pre)} & ci \in insts_{ct}(\sigma_1), ri \in roleinsts_r(\sigma_1), \\ & adoptedBy_r(\sigma_1)(ri) = ci, \end{array}$$

$$\begin{array}{ll} \textit{(post)} & insts_{ct}(\sigma_2) = \begin{cases} insts_{ct}(\sigma_1) \backslash \{ci\} & \text{, if } \nexists ri' \neq ri. \\ & \quad adoptedBy_r(\sigma_1)(ri') = ci \\ insts_{ct}(\sigma_1) & \text{, otherwise} \end{cases} \\ & insts_{ct'}(\sigma_2) = insts_{ct'}(\sigma_1) \text{ for all } ct' \neq ct, \\ & roleinsts_r(\sigma_2) = roleinsts_r(\sigma_1) \backslash \{ri\}, \\ & roleinsts_{r'}(\sigma_2) = roleinsts_{r'}(\sigma_1) \text{ for all } r' \neq r, \\ & adoptedBy(\sigma_2) = adoptedBy(\sigma_1)|_{roleinsts(\sigma_2)}, \\ & data(\sigma_2) = data(\sigma_1), \\ & roledata(\sigma_2) = roledata(\sigma_1)|_{roleinsts(\sigma_2)}, \\ & control(\sigma_2) = control(\sigma_1)|_{roleinsts(\sigma_2)}. \end{array}$$

The class of all ensemble automata for an ensemble structure Σ *is denoted by* $\mathbf{EAut}(\Sigma)$.

Fig. 9 shows an example of an ensemble automaton for the peer-2-peer network. The state σ_6 corresponds to the $\Sigma_{transfer}$-state σ in Fig. 8. The peer instance $p1$ starts the task by joining the ensemble as a *requester* which creates a new role instance *req* for the *requester*. Then $p2$ joins the ensemble in the role of a *router* and the role instance *rout*1 for the first router is created in σ_2. The role instance *req* then sends to the router *rout*1 a request for the address of the peer who stores file `"song.mp3"`. Since *rout*1 is currently adopted by $p2$ which does not store the requested file, another peer $p3$ needs to join the ensemble in state σ_4 as a router. It adopts the new role instance *rout*2. Now, *rout*1 forwards the request for the address to *rout*2 leading to state σ_5. The component $p3$ stores the file and therefore additionally adopts the role of a *provider* realized by the role instance *prov* which is depicted in Fig. 8. Afterwards, the component $p3$, in its role as a router *rout*2, sends its address to the forwarding router *rout*1 and then the component $p3$ abandons its role as a router leading to state σ_8. Another forwarding step transmits the address from *rout*1 to *req*. The requester *req* can now directly request the file from the provider *prov* who sends the content of the file back to the requester. At this point, the task is finished in state σ_{11}.

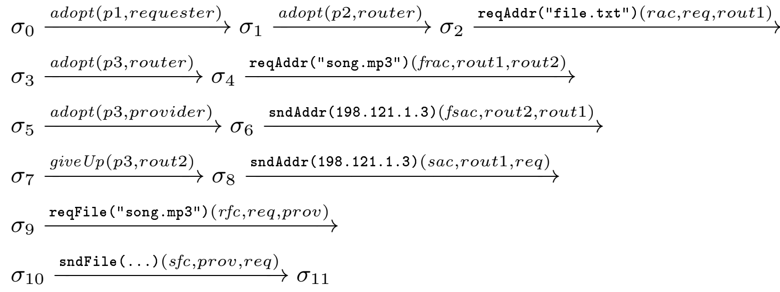


Fig. 9: Valid sequence of transitions in the $\Sigma_{transfer}$-model

In a Σ-ensemble automaton, the ensemble can behave arbitrarily as long as it uses legal transitions between Σ-ensemble states. However, we want role instances to act according to their specified role behaviors such that the ensemble works towards reaching a particular goal. These role behaviors restrict the Σ-ensemble automaton such that that only sequences of actions adhering to the behavior specifications of the roles are allowed. This leads to our notion of a *model of an ensemble specification.*.

Definition 12 (Model of an Ensemble Specification). *Let CT and INST be as in Def. 11. Let* $\Sigma = (roles, conns)$ *be an ensemble structure over CT, let* $EnsSpec = (\Sigma, RoleBeh)$ *with* $RoleBeh = (RoleBeh_r)_{r \in roles}$ *be an ensemble specification and let* $M = (S, \sigma_0, L, T)$ *with* $L = oplabels \cup mgmtlabels$ *be a* Σ*-ensemble automaton (over INST).*

M is a model of *EnsSpec, if the following conditions are satisfied:*

(1) for all $opname(actparams)(rc, ri, ri') \in oplabels, \sigma_1, \sigma_2 \in S$, *it holds:*
If $\sigma_1 \xrightarrow{opname(actparams)(rc,ri,ri')} \sigma_2 \in T$, *then*

(a) $control_{r''}(\sigma_1)(ri'') = control_{r''}(\sigma_2)(ri'')$ *for all* $r'' \in roles$, $ri'' \notin \{ri, ri'\}$,

(b) there exists $ri \in roleinsts_r(\sigma_1)$, $r \in roles$ *such that*
$control_r(\sigma_1)(ri) \xrightarrow{nm(rc).opname(params)!} control_r(\sigma_2)(ri) \in \Delta(RoleBeh_r)$, *and*

(c) it exists $ri' \in roleinsts_{r'}(\sigma_1)$, $r' \in roles$ *such that*
$control_{r'}(\sigma_1)(ri') \xrightarrow{nm(rc).opname(params)?} control_{r'}(\sigma_2)(ri') \in \Delta(RoleBeh_{r'})$

such that $actparams \in \mathcal{D}^*$ *is a list of actual parameters instantiating* $params$,

(2) for all $adopt(ci, r) \in mgmtlabels, \sigma_1, \sigma_2 \in S$, *it holds:*
If $\sigma_1 \xrightarrow{adopt(ci,r)} \sigma_2 \in T$ *with* $roleinsts_r(\sigma_2) = roleinsts_r(\sigma_1) \cup \{ri\}$, $ri \notin roleinsts_r(\sigma_1)$, *then* $control_r(\sigma_2)(ri) = q_0(RoleBeh_r)$.

The class of all models of EnsSpec is denoted by **Mod**(*EnsSpec*).

Condition (1a) says that control states of role instances that are not involved in the communication do not change. The rules (1b) and (1c) express that a communication between two role instances is only allowed if the role instances are in a control state of their respective role behaviors such that both roles are allowed to communicate. There are no restrictions on the particular instances that want to communicate since role behaviors are specified on the type and not on the instance level. Condition (2) requires that whenever a role instance is created its control state is the initial state of its role behavior. There are no particular constraints for the occurrence of management operations since those are not considered in role behaviors and therefore can always occur when their pre- and postconditions of Def. 11 are satisfied.

As an example, consider the ensemble specification in Sect. 2.3 with the three role behaviors specified in Fig. 5. The ensemble automaton shown in Fig. 9 respects the role behavior specifications and is therefore a model of the ensemble specification.

4 Related Work

Our framework is driven by a rigorous discrimination between instances and types. Formally, an ensemble is composed by a set of component instances such that each component instance, participating in the ensemble, adopts at least one role instance representing a role that the component currently plays in a collaboration. Of course, sets of interacting components are considered in any reasonable component model. They occur in the form of architectures [4,10,13],

networks [5], assemblies [11,22], team automata [8], etc. Mostly, components and their behaviors are described on the type level such that the dynamic creation of individual instances, their identification and the evolution of systems is not supported. Exceptions are component interaction automata [11], which identify components by names such that individual communications naming sender and receiver of a message are possible (similarly to communication between role instances in ensemble automata), and SCEL [18] which additionally allows dynamic creation of components. This is possible since SCEL considers two levels of operational semantics, the component level and the system level. Similarly, the HELENA approach distinguishes between role behaviors (on the type level) and ensemble behaviors (on the instance level). We do not create new component instances during the run of an ensemble because we assume them to be already given by an overall system management when an ensemble is started. However, component instances can dynamically join and leave an ensemble while role instances are dynamically created (and adopted by a component instance) during an ensemble execution. Also in the DEECo model [12] for ensemble-based component systems the membership of components in ensembles is dynamically changing which is realized by the DEECo runtime framework. Interaction of ensemble members is implicit in DEECo and performed via knowledge exchange triggered by the DEECo infrastructure. A computational model for DEECo is defined in terms of automata [3] that express knowledge exchange by buffered updating of components' knowledge. A general mathematical system model for ensembles based on input/output relations has been presented in [24]. It aims at general applicability such that, e.g., also physical parts based on differential equations can be integrated. HELENA is more concrete since at least explicit notions of interaction and collaboration (on the type and on the instance level) are involved.

In contrast to the other component models HELENA is centered around the notion of a role which allows to focus only on those capabilities of a component that is actually needed in a particular collaboration. The use of roles has already been proposed in [21,26] as an additional concept to classes and objects in object-oriented programming. In [26] it is stated that "a role of an object is a set of properties which are important for an object to be able to behave in a certain way expected by a set of other objects". In these approaches the consideration of role instances is already recommended and, in [26], a diagrammatic specification of role behaviors is suggested. Experiments with implementing roles in Smalltalk are also discussed. In [32] a formal definition for "model specifications" in the language LODWICK is proposed consisting of a signature, a static model and a dynamic model. The signature relates types and roles; the static model comprises all instances of types and their relationships to roles that may potentially exist; the dynamic model consist of sequences of sets of objects and their associated roles similar to state transitions in ensemble automata. LODWICK is designed as a rudimentary modeling language which does not contain collaboration specifications and does not support object interactions in the dynamic models. Apparently the ideas of role-based modeling did not have much influ-

ence on new methodologies for component-based systems engineering. Although UML2 has explicitly established a conceptual role layer between types and instances (for context dependent modeling), our impression is that its potential has not been sufficiently recognized yet.

The situation is different in the community of (multi-)agent systems where the modeling of roles is incorporated as a central part in methodologies for analysis and design. For instance, the GAIA methodology [35] and its extensions [15] consider a multi-agent system as a computational organization consisting of various interacting roles; this is very similar to our interpretation of ensembles. Most specifications in this methodologies are, however, rather informal or at most semi-formal, like the UML-based notation Agent UML [6]. Agent UML models collaborations by interaction protocols which combine sequence diagrams with state diagrams. Another approach has been pursued in the ROPE project [7], which proposes to use "cooperation processes" represented by Petri nets for the specification of collaborative behavior. A model-driven approach to the development of role-based open multi-agent software is presented in [36]. It uses Object-Z notation and focuses merely on structural properties of role organizations and agent societies and not on interaction behavior. The structural concepts involve, however, specifications of role spaces as containers of role instances (that can be taken by agents), which resembles ensemble states in HELENA. All these methods are not based on a formal semantics and do not provide verification techniques which will be a central topic of our approach in the near future. In particular, they do not formalize concurrent executions which is built-in in our ensemble automata expressed by interleaving.

5 Conclusion

In this paper, we presented the HELENA approach for modeling ensemble-based systems. HELENA extends the component-based approach by the notion of roles teaming up in ensemble to collaborate for some global goal. We introduced ensemble structures to capture the static architecture of such teams composed of roles and role connectors for communication between roles. For the dynamic aspects, an ensemble specification adds role behaviors to ensemble structures. The formal semantics and execution model of an ensemble specification was given as an ensemble automaton for the evolving ensemble with synchronous communication. We illustrated our approach by the running example of a peer-2-peer network for storing and downloading files.

We consider our work as a first step towards a comprehensive methodology for the development of ensemble systems founded on a precise semantic basis. We have not yet considered an infrastructure for the administration of ensembles. Several variants are possible dependent on the choice of a concrete interaction and/or communication model. For instance, we will define also an execution model for asynchronous communication which can be realized by message passing via event queues. A further important issue concerns the transition from ensemble specifications to designs and implementations. One possibility is to

use component-based architectures for the target systems and to map ensemble specifications (semi-automatically) to a component-based design for concurrent executions of ensembles. The mapping depends again on the choice of particular interaction models like synchronous and buffered communication including multicast message passing. We also plan to study an interaction model based on knowledge repositories and knowledge exchange like in SCEL and DEECo. Since SCEL can be considered as an abstract programming language we envisage to implement ensemble specifications by abstract SCEL programs. The semantic foundations of both languages should be appropriate to verify the correctness of the implementation. Another possibility is a direct implementation of ensemble specifications by using an appropriate framework, a prototype of which has currently been developed [25]. As a next step, we want to investigate under which conditions properties of communication compatibility (see e.g. [23]) valid for role behaviors can be propagated to ensemble automata and implementations. The challenge here is that role behavior specifications are formalized for types while ensemble automata (and implementations) concern concurrently executing instances.

Concerning the first phase of the development life cycle our methodology should still be augmented with explicit interaction specifications. Currently our behavioral descriptions are local to single role behaviors, but do not explicitly model the interactions to achieve a goal on a global level. For that purpose, we want to investigate appropriate notations, for instance on the basis of communication protocols used for specifying global interactions in multi-party sessions [14], [16]. The transition from an interaction specification to an ensemble specification must be formalized by an appropriate refinement relation. Then we want to consider properties of interaction specifications (expressed in some logic) and to prove that they are preserved by refinement. Also the explicit integration of adaptation and awareness requirements, which are central to autonomously evolving systems, must be considered. We need techniques to specify goals, for instance in the style of KAOS [27], and we need verification techniques for goal satisfaction. The validation of Helena w.r.t. the case studies of the ASCENS project (e-mobility, robotics rescue scenario, autonomic cloud platform) is currently ongoing.

Acknowledgment. We would like to thank Lenz Belzner, Giulio Iacobelli, Nora Koch and Rocco De Nicola for intensive and stimulating discussions and Philip Mayer for reading and commenting a draft version of this paper. We would also like to thank the anonymous reviewers for their constructive and helpful comments on the first version of this paper.

References

1. The ASCENS Project, `http://www.ascens-ist.eu`
2. Abeywickrama, D., Bicocchi, N., Zambonelli, F.: SOTA: Towards a General Model for Self-Adaptive Systems. In: 21st IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises. pp. 48–53. IEEE CS Press, Toulouse (2012)
3. Ali, R.A., Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: DEECo Computational Model - I. Tech. Rep. D3S-TR-2013-01, Charles University in Prague (2013)
4. Allen, R., Garlan, D.: A Formal Basis for Architectural Connection. ACM Trans. Softw. Eng. Methodol. 6(3), 213–249 (1997)
5. Barros, T., Ameur-Boulifa, R., Cansado, A., Henrio, L., Madelaine, E.: Behavioural models for distributed Fractal components. Annales des Télécommunications 64(1-2), 25–43 (2009)
6. Bauer, B., Müller, J.P., Odell, J.: Agent UML: A formalism for specifying multiagent software systems. Int. Journal of Software Engineering and Knowledge Engineering 11, 91–103 (2000)
7. Becht, M., Gurzki, T., Klarmann, J., Muscholl, M.: ROPE: Role Oriented Programming Environment for Multiagent Systems. In: Proceedings of the Fourth IECIS International Conference on Cooperative Information Systems. pp. 325–333. COOPIS '99, IEEE Computer Society, Washington, DC, USA (1999)
8. ter Beek, M.H., Ellis, C.A., Kleijn, J., Rozenberg, G.: Synchronizations in Team Automata for Groupware Systems. Computer Supported Cooperative Work 12(1), 21–69 (2003)
9. Bensalem, S., Bures, T., Combaz, J., De Nicola, R., Hölzl, M., Koch, N., Loreti, M., Tuma, P., Wirsing, M., , Zambonelli, F.: A Life Cycle for the Development of Autonomic Systems. 3rd Awareness Workshop at SASO 2013 - submitted (2013)
10. Bernardo, M., Ciancarini, P., Donatiello, L.: Architecting families of software systems with process algebras. ACM Trans. Softw. Eng. Methodol. 11(4), 386–426 (2002)
11. Brim, L., Černá, I., Vařeková, P., Zimmerova, B.: Component-interaction automata as a verification-oriented component-based system specification. SIGSOFT Softw. Eng. Notes 31(2), 4 (2006)
12. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: DEECo: an ensemble-based component system. In: Proceedings of CBSE 2013, Vancouver, Canada. pp. 81–90. ACM (2013)
13. Bures, T., Hnetynka, P., Plasil, F.: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In: SERA. pp. 40–48 (2006)
14. Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On global types and multiparty sessions. In: Proceedings of the joint 13th IFIP WG 6.1 and 30th IFIP WG 6.1 international conference on Formal techniques for distributed systems. pp. 1–28. FMOODS'11/FORTE'11, Springer-Verlag, Berlin, Heidelberg (2011)
15. Cernuzzi, L., Juan, T., Sterling, L., Zambonelli, F.: The Gaia Methodology: Basic Concepts and Extensions. In: Bergenti, F., Gleizes, M.P. (eds.) Methodologies and Software Engineering for Agent Systems, Multiagent Systems, Artificial Societies, and Simulated Organizations, vol. 11, pp. 69–88. Springer US (2004)
16. Coppo, M., Dezani-Ciancaglini, M., Padovani, L., Yoshida, N.: Inference of global progress properties for dynamically interleaved multiparty sessions. In: COORDINATION. pp. 45–59 (2013)

17. 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.) FMCO. Lecture Notes in Computer Science, vol. 7542, pp. 25–48. Springer (2011)
18. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: SCEL: a Language for Autonomic Computing. Tech. rep., IMT, Institute for Advanced Studies Lucca, Italy (2013)
19. Futatsugi, K., Goguen, J.A., Jouannaud, J.P., Meseguer, J.: Principles of OBJ2. In: Van Deusen, M.S., Galil, Z. (eds.) POPL. pp. 52–66. ACM Press (1985)
20. Goguen, J.A., Burstall, R.M.: Institutions: Abstract Model Theory for Specification and Programming. J. ACM 39(1), 95–146 (1992)
21. Gottlob, G., Schrefl, M., Röck, B.: Extending Object-Oriented Systems with Roles. ACM Trans. Inf. Syst. 14(3), 268–296 (1996)
22. Hennicker, R., Knapp, A.: Modal Interface Theories for Communication-Safe Component Assemblies. In: ICTAC. pp. 135–153 (2011)
23. Hennicker, R., Knapp, A.: Modal interface theories for communication-safe component assemblies. In: Cerone, A., Pihlajasaari, P. (eds.) ICTAC. Lecture Notes in Computer Science, vol. 6916, pp. 135–153. Springer (2011)
24. Hölzl, M.M., Wirsing, M.: Towards a System Model for Ensembles. In: Formal Modeling: Actors, Open Systems, Biological Systems. pp. 241–261 (2011)
25. Klarl, A., Hennicker, R.: The Helena Framework, http://www.pst.ifi.lmu.de/Personen/team/klarl/papers/helena.jar
26. Kristensen, B.B., Østerbye, K.: Roles: Conceptual Abstraction Theory and Practical Language Issues. TAPOS 2(3), 143–160 (1996)
27. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley (2009)
28. Mori, A., Futatsugi, K.: Verifying Behavioural Specifications in CafeOBJ Environment. In: Wing, J.M., Woodcock, J. (eds.) World Congress on Formal Methods. Lecture Notes in Computer Science, vol. 1709, pp. 1625–1643. Springer (1999)
29. Nakajima, S., Futatsugi, K.: An Object-Oriented Modeling Method for Algebraic Specifications in CafeOBJ. In: Adrion, W.R., Fuggetta, A., Taylor, R.N., Wasserman, A.I. (eds.) ICSE. pp. 34–44. ACM (1997)
30. OMG (Object Management Group): OMG Unified Modeling Language Superstructure. Specification, OMG (Object Management Group) (2011), http://www.omg.org/spec/UML/2.4.1/Superstructure/
31. Rausch, A., Reussner, R., Mirandola, R., Plasil, F. (eds.): The Common Component Modeling Example: Comparing Software Component Models, LNCS, vol. 5153. Springer (2008)
32. Steimann, F.: On the representation of roles in object-oriented and conceptual modelling. Data Knowl. Eng. 35(1), 83–106 (2000)
33. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edn. (2002)
34. Wirsing, M., Hölzl, M., Tribastone, M., Zambonelli, F.: ASCENS: Engineering Autonomic Service-Component Ensembles. In: Beckert, B., Damiani, F., Bonsangue, M., de Boer, F. (eds.) Formal Methods for Components and Objects, 10th International Symposium, FMCO 2011. LNCS, Springer (2012)
35. Wooldridge, M., Jennings, N.R., Kinny, D.: The Gaia Methodology for Agent-Oriented Analysis and Design. Autonomous Agents and Multi-Agent Systems 3(3), 285–312 (2000)
36. Xu, H., Zhang, X., Patel, R.J.: Developing Role-Based Open Multi-Agent Software Systems. International Journal of Computational Intelligence Theory and Practice 2 (2007)