Design and Implementation of Dynamically Evolving Ensembles with the HELENA Framework

Annabelle Klarl
Ludwig-Maximilians-Universität München
Munich, Germany
Email: klarl@pst.ifi.lmu.de

Rolf Hennicker
Ludwig-Maximilians-Universität München
Munich, Germany
Email: hennicke@pst.ifi.lmu.de

***Abstract*—Ensembles are collections of autonomic entities which collaborate to perform certain tasks. They show typically a complex dynamic behavior which is difficult to implement with state of the art development techniques. In this paper, we present a systematic methodology for the design and implementation of ensemble-based systems which goes beyond component-based development. A conceptual key point of our approach (elaborated in [1]) is that components can adopt different roles and that they can participate (under certain roles) in several, possibly concurrently running ensembles. In this paper, we present a novel developer framework that extends the component-based approach by explicitly taking into account roles and ensembles. The framework implementation follows rigorous rules formalized in terms of ensemble-structures and ensemble automata. Its application is demonstrated by a peer-2-peer file system network.**

I. Introduction

The vision of autonomic computing [2] drives the development of cutting-edge software systems. We no longer want to create systems which are thoroughly administrated by hand. Once employed, the system should rather manage itself, keep itself alive and running. At the same time, ubiquitous computing [3] and global interconnectedness introduce more and more nodes into our systems. They become highly distributed and need to operate in diverse and changing environments. Those systems challenge us to think of designs for autonomic systems with large numbers of autonomic nodes in many concurrently running teams, each fulfilling its global goal despite unforeseen and changing conditions. We even have to cope with changing compositions of teams while maintaining the overall functionality of the system.

The EU project ASCENS [4], [5] pursues the goal to develop foundations, techniques, and tools to construct those autonomic systems exhibiting complex interaction behaviors between highly dynamic and distributed nodes in diverse and changing environments. The basic building blocks are components which provide the main functionalities. These dynamically form *ensembles* to collaborate for some global goal. Following [6], [7], we distinguish between "achieve goals", meaning that the ensemble has to reach a desired state, and "maintenance goals", expressing that the ensemble needs to maintain a desired property throughout its execution. Well-known techniques, like component-based software engineering [8], [9], are not sufficient to model the dynamics and requirements of ensemble-based systems. Component systems describe the architecture of a target system. Ensembles are built on top of that as goal-oriented communication groups such that components dynamically adopt different responsibilities. Therefore, we need to be able to model that components can join and leave ensembles without breaking the overall functionality of the ensemble, that they can dynamically change their behavior according to their current *role* in an ensemble, and that they can even take part concurrently in different ensembles under ensemble-specific roles.

In the HELENA approach [1], we propose a modeling technique centered around the notion of roles teaming up in ensembles. The foundations of such ensemble-based systems are components which provide basic capabilities available in all roles the component can play. On top of that, we define *roles*, more precisely role types, which are able to take over responsibilities for a certain part of the ensemble task. Each role type must be supported by at least one component type whose instances are able to adopt that role. The role types add role-specific attributes and communication abilities. To define the structural characteristics of collaborations we use *ensemble structures*. They define which role types are needed in a collaboration and determine which role types may interact by which message types, specified by role connector types. This introduces a level of security since other interactions are not legal. Besides the structural architecture of ensembles, we specify the dynamic behavior of each role type with a labeled transition system. An *ensemble specification* defines an ensemble structure together with behavior specifications for all involved roles, thus determining the collaboration needed to solve a specific task. The HELENA approach also provides a formal semantics for ensemble specifications. It introduces an execution model such that the evolution of an ensemble is described by an *ensemble automaton*. The states of an ensemble automaton describe the currently existing role instances and which component instances currently adopt which roles. The transitions model either message exchange between role instances (along a connector) or management operations expressing that a component will either adopt or give up a certain role in the next ensemble state. Any ensemble automaton follows general preconditions for firing transitions and postconditions determining the effects of dynamically changing ensemble compositions.

In this paper, we present the architecture and implementation of a novel developer framework for the realization of ensemble-based systems following the HELENA modeling approach of [1] (the framework can be downloaded from [10]). The goal of the framework is, on the one hand, to implement the structural and dynamic rules enforced by both ensemble specifications and ensemble automata and, on the other hand, to facilitate the implementation of concrete ensemble systems. The framework contains two layers, a `metadata` layer and a `developer interface`. With help of the `metadata` layer, the user can define ensemble structures. For that, the `metadata` classes must be instantiated by objects which represent the various kinds of types that can occur in an ensemble structure, like role types, message types, etc. Thus an ensemble structure is represented by a net of objects which are linked in accordance with the general rules for ensemble structures. For instance, an ensemble structure can only contain role connector types which connect role types declared in the same ensemble structure. Restrictions like this are expressed by OCL constraints whose satisfaction is controlled by the framework.

The `developer interface` contains abstract base classes to implement concrete components, roles, messages, etc. They are related to the `metadata` classes by associations determining types. For instance, the developer interface contains an abstract class `Role` with an association to the `metadata` class `RoleType` such that any concrete role instance is associated with a unique (role) type. The abstract classes of the `developer interface` must be extended by the system developer to implement concrete ensembles in accordance with a particular ensemble structure (defined on the `metadata` level). Most importantly, for each concrete role class the behavior of the instances of that role must be realized. The framework prescribes that any role instance is an active object implemented as a thread whose run method executes the role behavior. Behavior implementations rely on message exchange between roles and management operations as specified in ensemble automata. For message exchange the framework offers connectors of appropriate types that establish physical connections between role instances. The framework controls that only connections allowed by the ensemble structure to which the role instances belong can be established. It also ensures that the rules for the transitions of ensemble automata are implemented in a correct way by respecting OCL pre- and postconditions derived from the pre- and postconditions of the transitions.

The development of both framework layers was driven by a systematic realization of the abstract concepts and rules of ensemble structures and ensemble automata. Additionally, the framework contains a system manager class which must be extended for a particular application such that specific ensemble structures can be instantiated and components and ensembles can be created and run, possibly concurrently. The system manager controls that first the components are created since they must be usable - under appropriate roles - in many ensembles. They can dynamically join and leave ensembles (by adopting or giving up particular roles); they can play different roles at the same time in the same ensemble, and they can also participate in several ensembles at the same time under various roles, provided that the role type is supported by the type of the component.

The paper is organized as follows: In Chapter II, we summarize the HELENA approach for ensemble modeling using ensemble specifications and ensemble automata. Chapter III presents our framework for implementing ensemble-based systems. Ensemble modeling and implementation are illustrated by a small case study. We finish with a discussion of related work and concluding remarks in Chapter IV.

II. ENSEMBLE MODELING

In this chapter, we give an overview the HELENA approach for modeling ensembles; cf. [1].

Notation 1: In the following, whenever we work with tuples $t = (t_1, \ldots, t_n)$, we use the notation $t_i(t)$ to refer to t_i.

A. 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. To classify components we use *component types*. A component type defines a set of attributes (more precisely attribute types) representing kernel information that is useful in all roles the component can adopt.[1] Formally, an *attribute type* is just a named variable 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.

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. To classify the roles that components can play we use role types. A role type determines first the types of the components that should be able to adopt this role. It also defines role-specific attribute types (to store data that is only relevant for performing the role) and it defines message types for those messages which are sent or received in order to fulfill the responsibilities of the role. Formally, a *message type* is of the form $msg = msgnm(params)$ such that $msgnm$ is the name of the message type and $params$ is a list of untyped formal parameters. 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,
- $rolemsgs = \langle rolemsgs_{out}, rolemsgs_{in} \rangle$ specifies types for outgoing and incoming messages supported by the role type.

To collaborate on tasks, roles need to communicate. A role can initiate information transfer via the call of an outgoing message and it can receive information via the reception of an incoming message. However, for the specification of collaborations we do not only want to declare communication abilities for each single role as given in the declaration of a role type, but we also want to specify which roles are meant to interact by which messages. This information is specified by a *role connector type* which is directed from a source role type to a target role type.

[1]We do not use component operations here, since we focus on message exchange between roles.

Given a set RT of role types, a *role connector type* over RT is a tuple $rct = (nm, srcType, trgType, msgs)$ such that

- nm is the name of the role connector type,
- $srcType \in RT$ denotes the source role type from which information is transferred along rct,
- $trgType \in RT$ denotes the target role type to which information is transferred along rct, and
- $msgs$ is a set of message types supported by rct such that $msgs \subseteq rolemsgs_{out}(srcType) \cap rolemsgs_{in}(trgType)$.

Role types and role connector types form the basic building blocks for collaborations in an ensemble. An *ensemble structure* determines the type of ensembles that are needed to perform a certain task. It specifies which roles must be available to contribute to the collaboration and which role connectors are required for interaction between those roles. Thus, an ensemble structure specifies the structural aspects of a collaborating ensemble.

Definition 1 (Ensemble Structure): Let CT be a set of component types. An *ensemble structure* Σ *over* CT is a pair $\Sigma = (roleTypes, rconnTypes)$ such that

- $roleTypes$ is a set of role types over CT such that each $rt \in roleTypes$ has a multiplicity $\mathit{mult}(rt) \in Mult$ and $Mult$ is the set of multiplicities available in UML, like $0..1$ or $*$,
- $rconnTypes$ is a set of role connector types over $roleTypes$ such that for each $rct \in rconnTypes$, it holds $srcType(rct), trgType(rct) \in roleTypes$.

The multiplicitiy associated to each role type specifies how many instances of that role type may (or must) participate in an ensemble. An ensemble structure Σ is *closed* if each message type supported by some role type of Σ is used in some role connector type of Σ. In this paper, we consider only closed ensemble structures.

After having modeled the structural aspects of ensembles, we focus on the specification of dynamic behaviors for each role type. Starting from an initial state, a role behavior specifies the sequences of messages exchanged by the role. For the specification of role behaviors, we use labeled transition systems whose labels express either sending (denoted by "!") or receiving (denoted by "?") a message along a role connector. Let Σ be an ensemble structure and $rt \in roleTypes(\Sigma)$. A *role behavior* of rt is given by a labeled transition system $RoleBeh_{rt} = (Q, q_0, \Lambda, \Delta)$ 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
  $\{rctnm.msg! \mid \exists rct \in rconnTypes(\Sigma) :$
  $\quad rctnm = nm(rct), rt = srcType(rct),$
  $\quad msg \in msgs(rct)\} \cup$
  $\{rctnm.msg? \mid \exists rct \in rconnTypes(\Sigma) :$
  $\quad rctnm = nm(rct), rt = trgType(rct),$
  $\quad msg \in msgs(rct)\}$,
- $\Delta \subseteq Q \times \Lambda \times Q$ is a transition relation.

The full specification of all structural and dynamic aspects of an ensemble determines the type of the collaboration in terms of an ensemble structure Σ and a set of all associated role behavior specifications, one for each role type.

Definition 2 (Ensemble specification): An *ensemble specification* is a pair $EnsSpec = (\Sigma, RoleBeh)$ such that

- Σ is an ensemble structure, and
- $RoleBeh = (RoleBeh_{rt})_{rt \in roleTypes(\Sigma)}$ is a family of role behaviors $RoleBeh_{rt}$.

B. Example: File Transfer Ensemble

We want to illustrate the specification of ensembles at a peer-2-peer network supporting the distributed storage of files which can be retrieved upon request. We start by defining the underlying component system comprising the basic component type `Peer`. A `Peer` has the attributes `address`, `fileNames`, and `contents`, the latter storing the contents of the files whose names are given by the attribute `fileNames`. For the visualization of the component type, we use a UML-like graphical notation (cf. Fig. 1). We assume that single components of type `Peer` are organized according to some predefined network topology, for instance as a component ring.

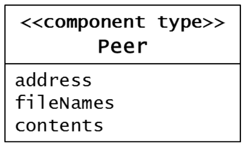


Figure 1: Component type `Peer`

In the second step, we specify the roles that peer components may adopt to perform a file transfer scenario. When a file is requested, several peers of the network collaborate within an ensemble. One peer plays the role of the requester of the file. Before downloading the file it must first figure out the address of the peer which stores the desired file. The search for the address is done by iterativeley forwarding the address request along the network structure. The peers that forward the address request (and send the answer back) play the role of a router. If a router determines that it has the requested file it will adopt the role of a provider. Finally, the requester downloads the file directly from the provider. We model the different roles with the role types `Requester`, `Router` and `Provider`. For their visualization we use again a UML-like notation with a new stereotype for modeling role types; cf. Fig. 2. After the name of the role type, the set of component types is listed, in this case just `Peer`, which are able to adopt the role. Afterwards, role-specific attributes are listed. Only the requester has a role-specific attribute for storing the name of the requested file, but they all use the kernel attributes of the component type `Peer` to access their stored files. When communicating, they interact with role-specific messages which we will explain in the following. A requester must be able to request the address of the provider from a router and receive the reply. For this purpose, we introduce the message types `reqAddr(fn)` and `sndAddr(addr)`. Messages of the former type are used as outgoing messages of a requester and incoming messages of a router while messages of the latter type are used as incoming messages of a requester and outgoing messages of a router.

When a requester got to know the address of the peer storing the file, it must be able to request the file from the provider and receive the content. For this purpose, we introduce the message types `reqFile(fn)` and `sndFile(cont)`, the former for outgoing messages of the requester and incoming messages of the provider and the latter for incoming messages of the requester and outgoing messages of the provider. Finally, a router must be able to forward address requests to another router and to receive replies from another router. Therefore, the router role uses messages of the type `reqAddr(fn)` also as outgoing messages and messages of the type `sndAddr(addr)` also as incoming messages.

<<role type>> Requester:{Peer}
fileName
out reqAddr(fn) **in** sndAddr(addr) **out** reqFile(fn) **in** sndFile(cont)

<<role type>> Router:{Peer}
in/out reqAddr(fn) **in/out** sndAddr(addr)

<<role type>> Provider:{Peer}
in reqFile(fn) **out** sndFile(cont)

Figure 2: Role types `Requester`, `Router` and `Provider`

To specify which message types can be exchanged between which role types, we introduce role connector types. For instance, messages of type `reqFile(fn)` can only be sent from a requester to a provider. This is modeled by the role connector type `ReqFileConn` which is graphically represented in Fig. 3. On the other hand, messages of type `reqAddr(fn)` can be sent from a requester to a router and also from a router to a router. Hence, we need two role connector types for this message, `ReqAddrConn` and `FwdReqAddrConn`, the former directed from `Requester` to `Router` and the latter directed from `Router` to `Router`. The other role connector types are found analogously.

<<role connector type>> ReqAddrConn
src Requester **trg** Router
reqAddr(fn)

<<role connector type>> FwdReqAddrConn
src Router **trg** Router
reqAddr(fn)

<<role connector type>> ReqFileConn
src Requester **trg** Provider
reqFile(fn)

Figure 3: Role connector types `ReqAddrConn`, `FwdReqAddrConn`, and `ReqFileConn`

In the next step, we compose role types and role connector types to define an ensemble structure $\Sigma_{transfer}$ for file transfer ensembles on top of peer-2-peer networks. All aforementioned role types participate in the ensemble structure, but each with a different multiplicity. There may be at most one component playing the role of a requester and one for the provider role. However, arbitrarily many components may be involved as routers. The aforementioned role connector types are employed to exchange messages. 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.

Lastly, we specify the dynamic behavior of the three roles of the ensemble structure $\Sigma_{transfer}$ by the role behaviors

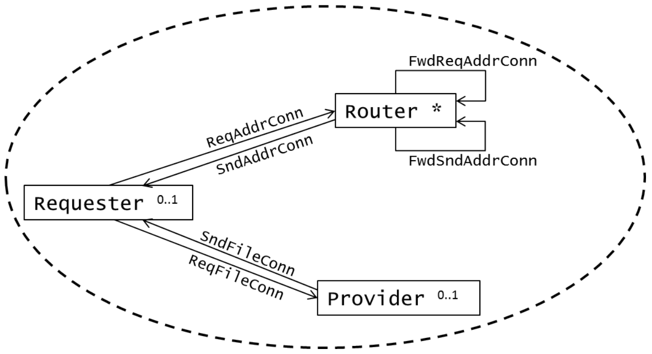


Figure 4: Ensemble structure $\Sigma_{transfer}$

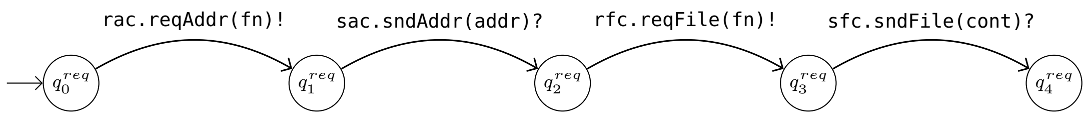
$RoleBeh_{\texttt{Requester}}$, $RoleBeh_{\texttt{Router}}$, and $RoleBeh_{\texttt{Provider}}$ shown in Fig. 5. In the figure, we use abbreviations for the role connector types; for instance `rac` stands for `ReqAddrConn` and `frac` for `FwdReqAddrConn`. All behaviors terminate in a final state since in this application we consider an achieve goal such that an ensemble stops when it has fulfilled its task. The specifications describe the required message exchange sequences of each role type. The behaviors of the roles `Requester` and `Provider` are specified as expected. However, a `Router` can first receive either a request address message from a requester, using the connector `rac`, or a request address message from (another) router, using the forward request address connector `frac`. In each case, when a request is received, the router has the choice to forward the request further, if it does not have the requested file, or to send its address back otherwise. This example illustrates that the separate consideration of roles facilitates significantly the specification of ensemble-based systems. If we would talk only on components, a complex component behavior specification would be needed subsuming all the different cases.

C. Ensemble Automata

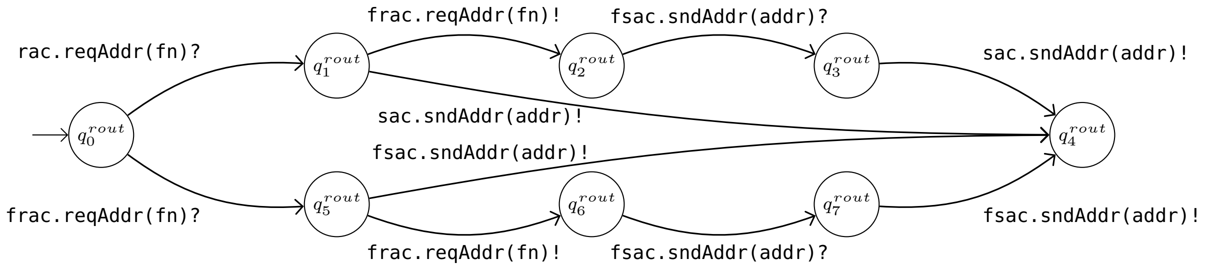
Now, we turn to an execution model for ensembles. Ensembles evolve over time by message exchange between collaborating roles or by performing management operations. For the formalization of system evolution, we use again labeled transition systems, but this time, in contrast to role behaviors, the states represent concrete ensemble states and the labels at the transitions represent either concrete messages (with actual parameters) exchanged by role instances or management operations.

Ensemble state. Given an ensemble structure Σ, an ensemble state σ has to record four kinds of information: 1) The currently existing 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 role instances. 4) The current control state of each role instance showing the current progress of the execution of its role behavior. An ensemble state σ represents this information as shown by the four items below. It is defined relative to a given set of component instances.[2]

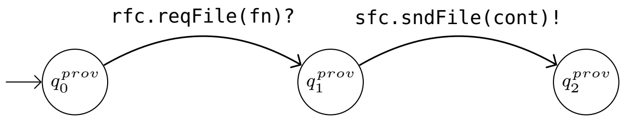
[2]Since component instances are of global nature and can be used across different ensembles, they do not belong themselves to an ensemble state.



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



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



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

Figure 5: Behavior specifications

Definition 3 (Ensemble state): Let Σ be an ensemble structure over a set CT of component types and let $INST = \bigcup_{ct \in CT}(INST_{ct})$ be the union of pairwise disjoint sets $INST_{ct}$ of component instances with type ct. A Σ-*ensemble state* (over $INST$) is a tuple $\sigma = (roleinsts, adoptedBy, roledata, control)$ such that:

1) $roleinsts = \bigcup_{rt \in roleTypes(\Sigma)}(roleinsts_{rt})$ is the union of pairwise disjoint sets $roleinsts_{rt}$ of role instances with type rt such that the multiplicities $mult(rt)$ are respected, e.g. $|roleinsts_{rt}| \leq 1$ if $mult(rt) = 0..1$.
2) $adoptedBy = (adoptedBy_{rt})_{rt \in roleTypes(\Sigma)}$ is a family of functions
$$adoptedBy_{rt} : roleinsts_{rt} \rightarrow \bigcup_{ct \in compTypes(rt)} INST_{ct}$$
3) $roledata = (roledata_{rt})_{rt \in roleTypes}$ is a family of functions $roledata_{rt} : roleinsts_{rt} \rightarrow [roleattrs(rt) \rightarrow \mathcal{D}]$, associating a mapping that assigns values in a universal data domain $\mathcal{D}$ to the attributes of each currently existing role instance of type rt.
4) $control = (control_{rt})_{rt \in roleTypes}$ is a family of functions $control_{rt} : roleinsts_{rt} \rightarrow CStates_{rt}$ such that $CStates_{rt}$ is a set of control states.

We say that a component instance $ci \in INST$ participates in an ensemble in a state σ if it is in the image of some $adoptedBy$ function. The set of all Σ-ensemble states is denoted by $States_{\Sigma}$.

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 $INST = INST_{\texttt{Peer}} = \{\texttt{p1}, \texttt{p2}, \texttt{p3}, \texttt{p4}\}$, 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. A graphical representation of this state is shown in Fig. 6. 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}.

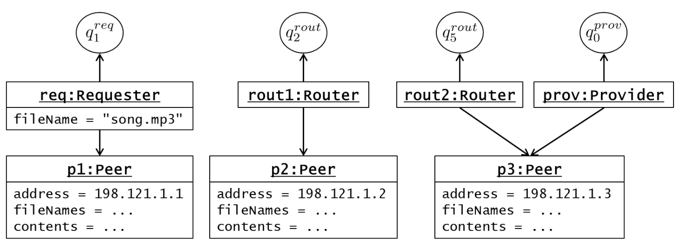


Figure 6: A $\Sigma_{transfer}$-ensemble state in graphical notation

Ensemble automaton. An ensemble automaton describes the execution of an ensemble in an abstract mathematical way. In [1], we have formally defined when an ensemble automaton conforms to a set of role behavior specifications. Here, we use ensemble automata as a semantic foundation for the development of the framework in Chapter III. The states of an ensemble automaton are ensemble states (as defined above) and the transitions are labeled either by message labels or by management labels. However, for the definition of an ensemble automaton, we must fix a communication style. In the following, we consider synchronous communication such that two communication partners must synchronize whenever they want to execute a shared input/output message. To express synchronous message exchange, we assume a set $msglabels$ of message labels of the form $msgnm(actparams)(rct, src, trg)$ meaning that a message with name $msgnm$ and actual parameters $actparams$ is sent, according to a role connector type rct, from a role instance *src* to a role instance *trg*. To express management actions, we assume a set *mgmtlabels* of management labels of the form $\mathit{createRole}(rt, ci)$ or $\mathit{giveUp}(ri, ci)$. The first label indicates that a role instance of type rt is created and adopted by the component instance ci. The second management label indicates that a role instance ri is given up by a component instance ci. In both cases, the *adoptedBy* function is updated accordingly. For all kinds of labels, appropriate constraints for pre- and/or poststates of transitions are provided. Preconditions specify applicability of an action, postconditions specify the effect of the action for an ensemble state; they are given informally, but can be easily formalized similarly to the preconditions. Missing preconditions express general applicability; missing postconditions and underspecified parts leave room for different implementations.

Definition 4 (Ensemble automaton): Let Σ, CT and $INST$ be as in Def. 3. A *Σ-ensemble automaton (over $INST$)* is a labeled transition system $M = (S, \sigma_0, L, T)$ such that

- $S \subseteq \mathit{States}_\Sigma$,
- $\sigma_0 \in S$ is the initial state,
- $L = \mathit{msglabels} \cup \mathit{mgmtlabels}$,
- for each $(\sigma_1, l, \sigma_2) \in T$, one of the following holds:
  - if $l = \mathit{msgnm}(\mathit{actparams})(rct, src, trg)$ then

    (pre) $rct \in \mathit{rconnTypes}(\Sigma)$, (1)

    $\mathit{msgnm}(\mathit{params}) \in \mathit{msgs}(rct)$, (2)

    actparams instantiates *params*, (3)

    $src \in \mathit{roleinsts}_{\mathit{srcType}(rct)}(\sigma_1)$, (4)

    $trg \in \mathit{roleinsts}_{\mathit{trgType}(rct)}(\sigma_1)$. (5)
  - if $l = \mathit{createRole}(rt, ci)$ then

    (pre) $rt \in \mathit{roleTypes}(\Sigma), ci \in \mathit{INST}_{ct}$, $ct \in \mathit{compTypes}(rt)$,

    (post) new role instance of type rt is created, and adopted by ci.
  - if $l = \mathit{giveUp}(ri, ci)$ then

    (post) role instance ri is removed, and not further adopted by ci.

III. Ensemble Implementation

In this chapter, we present our framework for implementing ensemble specifications (in Java) that are constructed following the HELENA modeling method described in Chapter II. The overall architecture of the framework is shown in Fig. 7. It contains two layers, the metadata layer and the developer interface which both are used by a system manager. The metadata layer allows us to define component types and ensemble structures in accordance with the definitions in Sect. II-A. The developer interface provides the basic functionality to implement concrete components, indicated by `C1,C2,C3`, and concrete ensembles, indicated by `E1,E2,E3,E4`, by extending the abstract base classes of the developer interface. It allows to realize role behaviors described in an ensemble specification (cf. Def. 2) and forces implementations to follow the abstract principles of an ensemble automaton given in Def. 4. The system manager and its concrete (application dependent) extension serve for the configuration of particular ensemble structures and for creating concrete ensembles which can run concurrently on top of a component-based system. Let us note that in both layers of the framework, the ensemble related parts are built upon the component related parts as indicated by the dependency arrows.

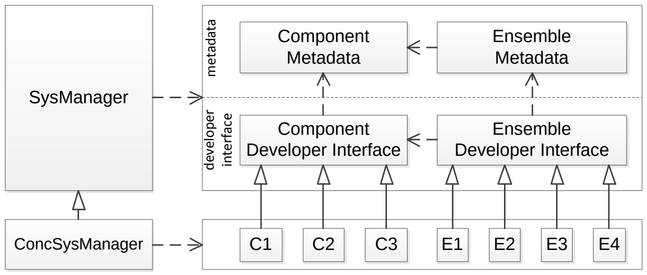


Figure 7: Architecture of the HELENA framework

A. Metadata Layer

All concepts used to build up ensemble structures are realized by corresponding metadata classes and the relationships between concepts are represented by associations in the `metadata` layer of the framework. The upper package of Fig. 8 gives an overview of this layer. For instance, to represent role types $rt = (nm, \mathit{compTypes}, \mathit{roleattrs}, \mathit{rolemsgs})$ we use a class `RoleType`. The name nm is stored in an attribute `name` of the class `RoleType` (not shown in the diagram). It has the type `Class<? extends Role>`. This ensures, using the reflection mechanism of Java, that only those objects of the class `RoleType` can be created whose `name` attribute refers to a role class extending the abstract class `Role` of the developer interface; see below. The set *compTypes* of component types which are able to adopt the role type is represented by an association with end `compTypes` directed from `RoleType` to the class `ComponentType`. The role type attributes *roleattrs* are determined by the association with end `roleattrTypes` directed from `RoleType` to the class `AttributeType`. Similarly, the set of message types $\mathit{rolemsgs}_{out}$ and $\mathit{rolemsgs}_{in}$ occurring in *rolemsgs* are modeled as associations directed to the class `MessageType`. Particular role types used in an ensemble structure are then represented by objects of the class `RoleType`. They are constructed with a static factory method `createType(...)` with parameters pointing to objects that represent the single constituent parts of a role type.

An ensemble structure $\Sigma = (\mathit{roleTypes}, \mathit{rconnTypes})$ is represented by an object of the class `EnsembleStructure` which has associations with ends `roleTypes` and `rconnTypes` to navigate to the role and role connector types of the ensemble structure. Similarly, the other concepts are realized in the metadata layer. In some cases there are structural restrictions involved in the definitions of Sect. II-A which cannot be expressed solely by the metadata classes and their associations. In these cases we use OCL invariants which are checked by the metadata layer when objects of the metadata classes are created. For instance, the following invariant expresses the second condition of Def. 1 which requires that for each role connector type occurring in an ensemble structure its source and target types must belong to the role types of the structure.

```
context EnsembleStructure inv:
   self.rconnTypes.allInstances()->forAll{rct|
     self.roleTypes->includes(rct.srcType) and
     self.roleTypes->includes(rct.trgType)}
```

Similarly we can express for role connector types $rct = (nm, srcType, trgType, msgs)$ the condition $msgs \subseteq rolemsgs_{out}(srcType) \cap rolemsgs_{in}(trgType)$ by an OCL invariant for the class `RoleConnectorType`.

B. Developer Interface

The goal of the developer interface is to facilitate the implementation of concrete ensemble applications by following the execution model of an ensemble automaton. In contrast to the `metadata` classes, we now concentrate on classes which can be extended such that instances represent concrete components, ensembles, roles, role connectors and messages. For this purpose the `developer interface` offers abstract classes `Component`, `Ensemble`, `Role`, etc. for each `metadata` class, apart from `AttributeType`.[3] An overview of the developer interface is shown in the second layer of Fig. 8. Each abstract class has an association to the corresponding metadata class such that the `type` of each instance of a concrete subclass can be determined. To implement a concrete ensemble application the abstract classes of the `developer interface` must be extended by concrete subclasses as indicated by the inheritance arrows in Fig. 8. The framework ensures, using Java reflection, that concrete subclasses and the attributes of concrete component and role classes fit to an ensemble structure represented by type instances on the `metadata` level.

The design of the developer framework is guided by the idea that any running ensemble can be abstractly considered as an ensemble automaton. Hence the states of a concrete ensemble should fit to the shape of an ensemble state as defined in Def. 3 and the execution steps should be relatable to the transitions of an ensemble automaton; see Def. 4. Let us first consider Σ-ensemble states which are tuples $\sigma = (roleinsts, adoptedBy, roledata, control)$. According to the developer framework an ensemble is represented by an instance of the class `Ensemble` which acts as an ensemble manager to execute management operations. The set of role instances $roleinsts$ currently existing in the ensemble is given by the association with end `roleinsts` directed from `Ensemble` to the class `Role`. For each role instance, the association `adoptedBy` navigates to the unique component instance which currently adopts this role, represented by the association end `owner`. Hence this association delivers us the information described by the $adoptedBy$ functions. The data state of each role instance, formalized by the functions in $roledata$, is given by the current values of the instance variables implementing the role attributes of concrete role classes. The current control state of each role instance, formalized by the $control$ functions, is implicitly given by the program counter reporting the progress of the execution of the role instance behavior. Moreover, an ensemble instance administrates a set of role connector instances using the association with end `rconninsts` from `Ensemble` to the class `RoleConnector`. This is an extension of the abstract notion of an ensemble state which is technically needed to establish communication. The ensemble automata considered here formalize synchronous communication. This is realized in the framework by the concrete subclass `SynchronousRoleConnector`. Other kinds of communication will be implemented in the future.

[3]We do not need an abstract class `Attribute` since attribute instances are implicitly represented by Java instance variables and their values associated to component and role instances.

For the realization of a role behavior the class `Role` prescribes the implementation of its abstract method `roleBehavior()`. The class `Role` extends the class `Thread` such that whenever a role instance is created a new thread is started which executes the `roleBehavior()` method (usually concurrently to other role instances). Additionally the role instance is supplied with an input channel.

According to the definition of ensemble automata, two types of labels advance the system: message labels $msglabels$ and management labels $mgmtlabels$. Message labels are of the form $msgnm(actparams)(rct, src, trg)$. The message part $msgnm(actparams)$ is represented in the framework by an instance of (a subclass of) `Message` while message parameters are implemented as attributes. The connector part (rct, src, trg) is represented by an instance of (a subclass of) `RoleConnector` which carries the information of the role connector type rct (stored in `type`) as well as the references to the source and target role instances src and trg stored in `src`, `trg` resp.. In the framework, message exchange is implemented by the use of the methods `send(msg:Message,rc:RoleConnector)` and `receive(expectedMsgTypes:ExpectedMsgType[])` of the class `Role`. The framework ensures that OCL preconditions, expressing the preconditions (1) - (5) stated in the ensemble automaton, are respected when a message is sent. Of course, before sending a message the source role instance must have created a role connector instance by calling the method `createRoleConnector(rcType:Class,trg:Role)` which returns, if admissible, a role connector of type `rcType` from `this` to `trg`. Hidden from the user, the role connector instance takes care to transmit the message to the `RoleInputChannel` of the target role instance where the target can receive the message. For synchronous message exchange we use an instance of `SynchronousInputQueue` which blocks the sender until the message is taken from the queue. The receive method has a parameter `expectedMsgTypes` determining which messages are currently expected as input. The framework checks at runtime whether the input requirements are met.

The management labels $createRole(rt, ci)$ and $giveUp(ri, ci)$ are implemented by calls to the corresponding methods in the class `Ensemble` (parameters are not shown). The framework ensures that the pre- and/or postconditions for the corresponding labels in Def. 4 hold, e.g. adopting a specific role type is allowed for the component. In this case the `createRole` method creates a new role instance and starts its thread as explained above. Both methods can be called by a role instance on its ensemble manager within the role behavior implementation. For instance, the behavior of a router who has detected that it stores the requested file calls `this.ens.createRole(Provider.class,this.owner);` and later on `this.ens.giveUp(this,this.owner);`. Sometimes, before creating a role instance, it is first necessary to find a suitable component instance that can adopt this role. For this purpose, the method

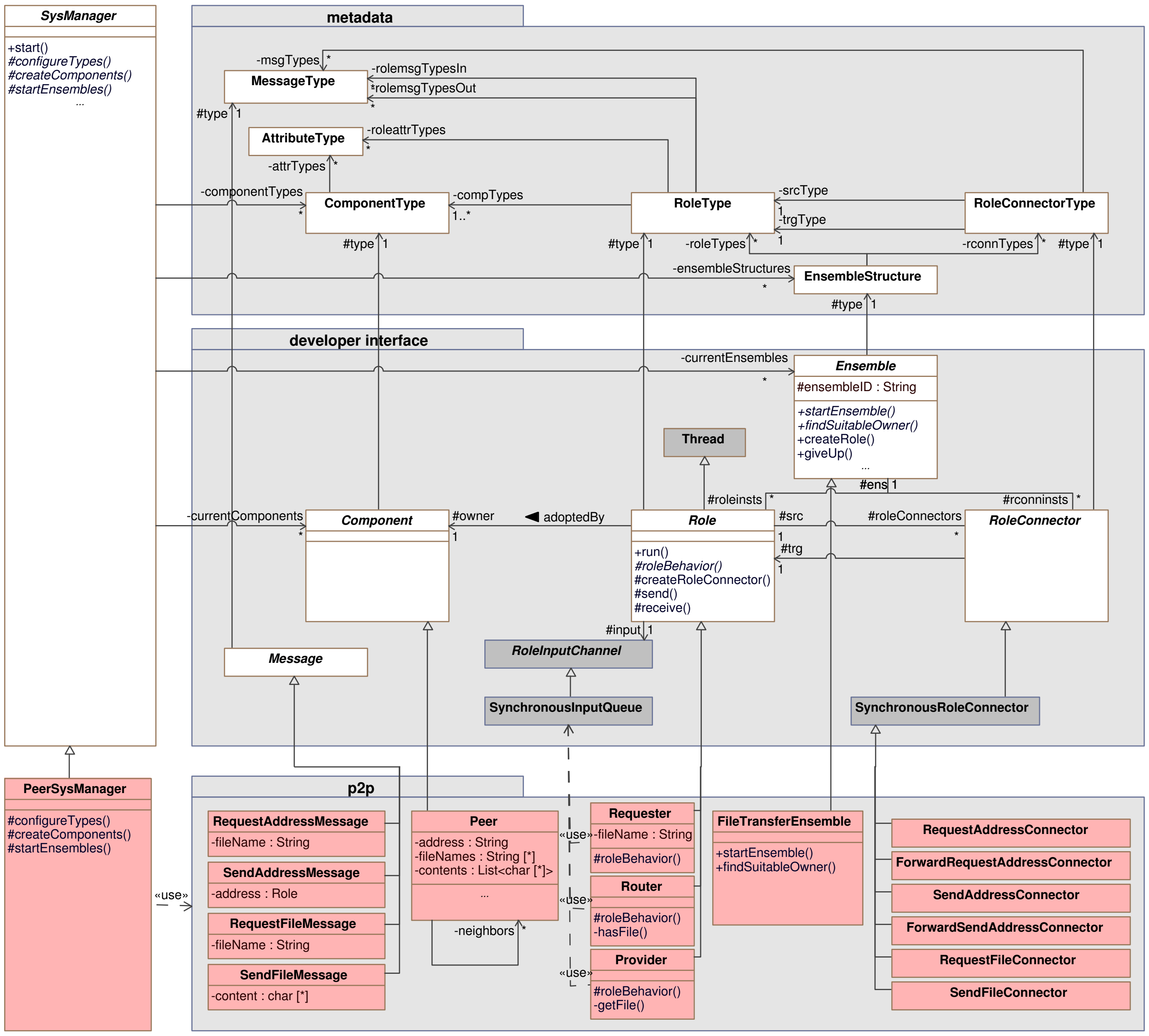


Figure 8: HELENA framework and application

`findSuitableOwner(roleType:Class,caller:Role)` of the class `Ensemble` can be used which has to be implemented in a concrete ensemble subclass.

It remains to mention the abstract `SysManager` class which provides a template method to `start` an ensemble system. This method calls sequentially the methods `configureTypes()` (to construct ensemble structures), `createComponents()` (to create the underlying component instances) and `startEnsembles()` to be implemented in the manager subclass of the framework application.

C. Framework Application

We want to illustrate the use of the framework by implementing the file transfer ensemble introduced in Sec. II-B. We perform the implementation in two steps concerning the static aspects and the dynamic behavior. For the static aspects, we firstly extend the superclasses of the `developer interface` for each type in the example as shown in Fig. 8: `Peer` extends `Component`, `Requester`, `Router`, and `Provider` extend `Role`, etc. We define attributes of components and roles, and parameters of messages, but we do not realize the role behaviors yet. Then, we extend the abstract superclass `SysManager` by the class `PeerSysManager` and implement the method `configureTypes()`. This method instantiates all type classes of the `metadata` layer and connects them appropriately to represent the ensemble structure $\Sigma_{transfer}$. We start from establishing the component types underlying the ensemble-based system (cf. line 2-4 in Fig. 9). For our example, we instantiate only one component type for peers (instantiation of attribute types for the peer type is not shown) and add that to the list `componentTypes` of the system manager. Afterwards, we create instances for all types of the ensemble structure and connect them accordingly. Line 6-11 exemplify this for the role type of a requester. Lastly, we compose all types to the desired ensemble structure and add it to the list of ensemble structures `ensembleStructures` for the system (line 14-15).

```
1  public void configureTypes() {
2    ComponentType peer =
3      ComponentType.createType(Peer.class, ...);
4    this.addComponentType(peer);
5
6    Set<ComponentType> reqCompTypes = getAsSet(peer);
7    Set<AttributeType> reqAttrTypes = ...;
8    Set<MessageType> reqMsgsOut = ...;
9    Set<MessageType> reqMsgsIn = ...;
10   RoleType req = RoleType.createType(Requester.class,
11     reqCompTypes, reqAttrTypes, reqMsgsOut,reqMsgsIn );
12   ...
13
14   this.addEnsembleStructure(EnsembleStructure.
15     createType(FileTransferEnsemble.class, ...);
16 }
```

Figure 9: Instantiation of types in method `configureTypes()`

The second step is to add dynamic behavior. For this purpose, we first realize the ensemble specification by a) implementing the methods `roleBehavior()` of all concrete role classes, b) implementing `findSuitableOwner()` of class `FileTransferEnsemble`, and c) implementing `startEnsemble()` of class `FileTransferEnsemble`. Then, we realize a concrete application by implementing `createComponents()` and `startEnsembles()` of the class `PeerSysManager`.

Implementing the method `roleBehavior()` for each concrete role class essentially means deriving an appropriate branching sequence of message exchanges from the labeled transition system for the role behaviors shown in Fig. 5. Each transition labeled with an output message is translated into a call of the method `send()` with appropriate parameters, and for input messages the method `receive()` is called respectively. Fig. 10 shows the implementation of $RoleBeh_{\texttt{Requester}}$ depicted in Fig. 5. However, different from the abstract behavior specification, the implementation needs to physically connect to the communication partner. For example for the first output message, the requester has first to get a reference to a router in the ensemble (line 2-3) and establish a `RequestAddressConnector` (line 5-7) before it can actually send the `RequestAddressMessage` (line 9). In the implementation of the method `findSuitableOwner()` (not shown), the ensemble manager retrieves the reference to the requested role by asking one of the neighboring components of the calling role instance to adopt the role.

The method `startEnsemble(Component initComp)` of the class `FileTransferEnsemble` actually starts an instance of the ensemble (cf. Fig. 11). The method gets an initial component as input where the file was initially requested. It creates a role instance of type `Requester` adopted by the initial (peer) component, thus starting to execute the requester's behavior. The name of the requested file is set on the role attribute of the requester in line 3.[4]

[4]In fact, the requester waits for this file name before it actually starts its behavior; the waiting process is not shown in Fig. 10.

```
1  protected synchronized void roleBehavior() {
2    Role router = this.ens.createRole(Router.class,
3     this.ens.findSuitableOwner(Router.class, this));
4
5    RequestAddressConnector rac =
6     this.createRoleConnector(
7       RequestAddressConnector.class, router);
8
9    this.send(new RequestAddressMessage(...), rac);
10
11   SendAddressMessage sndAddr =
12    (SendAddressMessage) this.receive(...);
13
14   RequestFileConnector rfc =
15    this.createRoleConnector(
16      RequestFileConnector.class, sndAddr.getAddress());
17
18   this.send(
19    new RequestFileMessage(this,
20    this.getAttribute("fileName",String.class)), rfc);
21
22   SendFileMessage sndFileMsg =
23    (SendFileMessage) this.receive(...);
24 }
```

Figure 10: Implementation of $RoleBeh_{\texttt{Requester}}$

```
1  public void startEnsemble(Component initComp) {
2    Requester req =
3     this.createRole(Requester.class,initComp);
4    req.setFileName(this.requestedFileName);
5  }
```

Figure 11: Implementation of `startEnsemble()`

Lastly, a concrete scenario needs to be set up. The system is populated by concrete peers in the method `createComponents()` of the `PeerSysManager` (cf. Fig. 12). Each peer is initialized as indicated in line 2-3, the network of peers as a ring structure is set up (line 5), and each peer is added to the list `currentComponents` (line 7). Afterwards, concrete ensemble instances are created and run in the method `startEnsembles()` (cf. 13).

```
1  public void createComponents() {
2    Peer peer1 = new Peer("p1", "192.121.1.1",
3                          fileNames, contents);
4    ...
5    peer1.addNeighbors(peer3, peer2);
6    ...
7    this.addComponent(peer1);
8    ...
9  }
```

Figure 12: Instantiation of peers in `createComponents()`

```
1  public void startEnsembles() {
2    Ensemble ens1 =
3     new FileTransferEnsemble("ens1", "song.mp3");
4    this.addEnsemble(ens1);
5    ens1.startEnsemble(this.getComponent(0));
6
7    Ensemble ens2 = ...
8  }
```

Figure 13: Implementation of `startEnsembles()`

Using our framework, the implementation of the case study was straightforward and could easily be derived from the formalization in HELENA. Different file transfer ensembles could be instantiated (cf. line 7 in Fig. 13) and run concurrently.

IV. CONCLUSION

We have provided a framework for the implementation of ensemble-based systems. The construction of the framework was guided by the abstract notions of ensemble specifications and ensemble automata used for modeling ensembles in the HELENA approach. HELENA extends component-based systems by the notion of roles and ensembles to focus on capabilities of a component needed for particular collaborations. Our framework transfers this concept to object-oriented programming by providing appropriate classes that can be extended for particular applications.

The idea to complement object-oriented systems with roles for evolving objects has already been introduced by Gottlob et al. [11] and Kristensen et al. [12]. A role provides a particular perspective on an object and is implemented as an adjunct instance to an object. However, they do not consider any dynamic behavior of roles or collaboration between roles to perform cooperative tasks. Steimann [13], [14] proposes a formal model for roles and relationships between roles embedding it into the role-oriented modeling language LODWICK. His "model specifications" are comprised of signature, static model, and dynamic model similarly to HELENA, but they do not specify any collaborations or object interactions. He proposes to indicate by interface realization which (component) types can adopt which roles. Hence, roles correspond to interfaces and do not provide behavior implementations. To describe structures of interacting objects (i.e. ensemble structures in HELENA) without having to take the entire system into consideration, Herrmann [15] introduces "teams" in his framework ObjectTeams/Java, Baldoni et al. [16] "institutions" in their framework powerJava, and Reenskaug [17] and Andersen [18] "role models" in their OOram method. Like in HELENA, they define the static architecture of a collaboration by participating roles, but they handle behavior very differently. In ObjectTeams/Java and powerJava, collaboration between roles is initiated through operation calls while in the OOram method, roles exchange message like in HELENA. In ObjectTeams/Java and powerJava, roles are not active themselves, but can only react to operation calls. Reenskaug and Andersen pursue our idea of roles as being autonomic entities which start their behavior based on an external stimulus (like a file being requested from the outside). However, while in HELENA we model concurrently running ensembles, Andersen proposes to compose overlapping role models into a single composite role model. He therefore suggests state spaces to efficiently represent composite behaviors while we explicitly want to run behaviors in parallel.

Our framework implementing the HELENA approach can be considered as a first prototype which will be extended in several directions: a) We will implement further connector types to support further communication styles like asynchronous communication and multicasting. b) The framework will be based on a component infrastructure supporting distributed deployment of components. c) A tool for the semi-automatic generation of role behavior implementations from role behavior specifications will be constructed. Moreover, the whole HELENA approach will be further developed to integrate goal specifications, awareness of changing environments, adaptability requirements and interaction specifications on a global level, like in multi-party session types [19]. We also want to investigate proof methods for checking properties on the level of ensemble specifications which should be preserved by implementations.

ACKNOWLEDGMENT

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

REFERENCES

[1] R. Hennicker and A. Klarl, "Foundations for Ensemble Modeling - The Helena Approach," in *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi (SAS 2014)*. LNCS, to appear 2014, preliminary version available at http://www.pst.ifi.lmu.de/Personen/team/klarl/papers/sas2014.pdf.

[2] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[3] M. Weiser, "The computer for the 21st century," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 3, no. 3, pp. 3–11, Jul. 1999.

[4] "The ASCENS Project." [Online]. Available: http://www.ascens-ist.eu

[5] M. Wirsing, M. Hölzl, M. Tribastone, and F. Zambonelli, "ASCENS: Engineering Autonomic Service-Component Ensembles," in *Formal Methods for Components and Objects, 10th International Symposium, FMCO 2011*, ser. LNCS. Springer.

[6] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[7] D. Abeywickrama, N. Bicocchi, and F. Zambonelli, "SOTA: Towards a General Model for Self-Adaptive Systems," in *21st IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*. Toulouse: IEEE CS Press, 2012, pp. 48–53.

[8] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[9] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, Eds., *The Common Component Modeling Example: Comparing Software Component Models*, ser. LNCS, vol. 5153. Springer, 2008.

[10] "The Helena Framework." [Online]. Available: http://www.pst.ifi.lmu.de/Personen/team/klarl/helena

[11] G. Gottlob, M. Schrefl, and B. Röck, "Extending Object-Oriented Systems with Roles," *ACM Trans. Inf. Syst.*, vol. 14, no. 3, pp. 268–296, 1996.

[12] B. B. Kristensen and K. Østerbye, "Roles: Conceptual Abstraction Theory and Practical Language Issues," *TAPOS*, vol. 2, no. 3, pp. 143–160, 1996.

[13] F. Steimann, "On the representation of roles in object-oriented and conceptual modelling," *Data Knowl. Eng.*, vol. 35, no. 1, pp. 83–106, 2000.

[14] ——, " Formale Modellierung mit Rollen," Habilitation Thesis, Universität Hannover, 2000.

[15] S. Herrmann, "Object teams: Improving modularity for crosscutting collaborations," in *Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, ser. NODe '02. London, UK, UK: Springer-Verlag, 2003, pp. 248–264.

[16] M. Baldoni, U. Studi, and T. Italy, "Interaction between Objects in powerJava," *Journal of Object Technology*, vol. 6, pp. 7–12, 2007.

[17] T. Reenskaug, *Working with objects: the OOram Framework Design Principles*. Manning Publications, Greenwich, CT, 1996.

[18] E. P. Andersen, " Conceptual Modeling of Objects – A Role Modeling Approach," Ph.D. dissertation, University of Oslo, 1997.

[19] M. Coppo, M. Dezani-Ciancaglini, L. Padovani, and N. Yoshida, "Inference of global progress properties for dynamically interleaved multiparty sessions," in *COORDINATION*, 2013, pp. 45–59.