From HELENA Ensemble Specifications to Executable Code

Annabelle Klarl, Lucia Cichella, and Rolf Hennicker

Ludwig-Maximilians-Universität München, Germany**

Abstract. The HELENA approach [5] provides a modeling technique for distributed systems where components dynamically collaborate in ensembles. Models of such systems are formalized with ensemble specifications. They can be implemented using the jHELENA framework [6]. In this paper, we present a domain-specific language for ensemble specifications and provide an Eclipse plug-in featuring an editor and an automatic code generator for translating ensemble specifications into executable code.

1 Motivation

Exploiting global interconnectedness in distributed systems, autonomic components can dynamically form ensembles to collaborate for some global goal. The EU project ASCENS [1,9] develops foundations, techniques and tools to support the whole life cycle for the construction of Autonomic Service Component ENSembles. In this context, several approaches to formalize and implement ensemble-based systems have been developed. SCEL [3,4] provides a kernel language for abstract programming of autonomic systems, whose components rely on knowledge repositories, and models interaction by knowledge exchange. In SCEL (and its implementation jRESP) ensembles are understood as group communications. DEECo [2] introduces an explicit specification artifact for ensembles dynamically formed according to a given membership predicate. Interaction is realized by implicit knowledge exchange managed by DECCo's runtime infrastructure. Related approaches have been developed in the context of multi-agent systems and multi-party session types, for instance in the Scribble framework [10]. Recently, we proposed the HELENA approach [5] which is centered around the notion of *roles*. Roles can be adopted by components to collaborate in ensembles. The introduction of roles helps (1) to focus on the particular tasks which components fulfill in specific collaborations and (2) to structure the implementation of ensemble-based systems. In the jHELENA framework [6], roles are implemented as Java threads on top of a component. Role objects are bound to specific ensembles while components can adopt many roles in different, concurrently running ensembles. So far, there is no tool support for writing ensemble specifications and their implementation in jHELENA must be derived by hand. In this paper, we present HELENATEXT, a domain-specific language for ensemble specifications, and provide an Eclipse plug-in for writing specifications and generating code following the strategy proposed in [6].

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

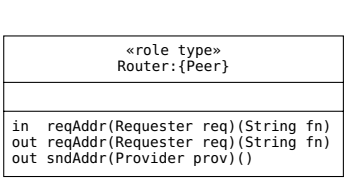
2 HELENA in a Nutshell

HELENA is based on a rigorous typing discipline, distinguishing between types and instances. Component instances classified by *component types* are considered as carriers of basic information relevant across many ensembles. Whenever a component instance joins an ensemble, the component adopts a role by creating a new role instance and assigning it to itself. The kind of roles a component is allowed to adopt is determined by role types. Given a set CT of component types, a *role type* rt over CT is a tuple $rt = (nm, compTypes, roleattrs, rolemsgs)$ such that nm is the name of the role type, $compTypes \subseteq CT$ is a finite, non-empty subset of component types (whose instances can adopt the role), *roleattrs* is a set of role specific attribute types for role-specific information, and *rolemsgs* is a set of message types capturing incoming, outgoing, and internal messages supported by the role type rt. We want to illustrate the use of HELENA at a peer-2-peer network supporting the distributed storage of files which can be retrieved upon request. Several peers work together to request and transfer a file: One peer plays the role of the `Requester` of the file, other peers act as `Router`s and the peer storing the requested file adopts the role of the `Provider`. All these roles can be adopted by components of the type `Peer`. Fig. 1a shows the role type `Router` in graphical representation similar to a UML class. The notation `Router:{Peer}` indicates that any component instance of type `Peer` can adopt the role `Router`. The `Router` has no role-specific attributes and supports one incoming and two outgoing messages types. The full specification and implementation of the example can be found in [5,6].

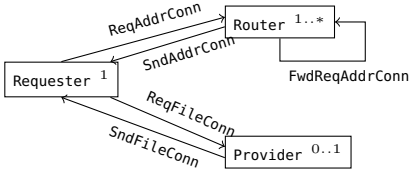
A HELENA *ensemble specification* $EnsSpec = (\Sigma, RoleBeh)$ consists of two parts, an *ensemble structure* Σ and a family $RoleBeh$ of *role behavior specifications* $RoleBeh_{rt}$ (one for each role type rt occurring in Σ). The ensemble structure $\Sigma = (roleTypes, rconnTypes)$ specifies a set $roleTypes$ of pairs, consisting of a role type and an associated multiplicity. Each multiplicity (like 0..1, 1, $*$, 1..$*$ etc.) determines how many instances of that role type may contribute to the ensemble. The set $rconnTypes$ of *role connector types* specifies which types of messages can be exchanged between role instances. Each role connector type must be equipped with a source and a target role type which must be declared in $roleTypes$. Fig. 1b shows a graphical representation of the ensemble structure for the p2p example. It consists of three role types (`Requester`, `Router`, `Provider`) with associated multiplicities and five role connector types. For instance, the connector type `ReqAddrConn` consists of the single message type `reqAddr(Requester req)(String fn)` with source type `Requester` and with target type `Router`. It will be used for requesting the address of a provider for file `fn` such that the file can be directly downloaded afterwards using the connectors between `Requester` and `Provider`.

A *role behavior specification* $RoleBeh_{rt}$ for a role type rt specifies the life cycle of each instance of rt. We represent role behaviors by labeled transition systems derived from process expressions [8]. The labels denote actions which must fit to the declared ensemble structure. There are actions for creating role instances, sending (!) or receiving (?) messages, and performing internal computations. For

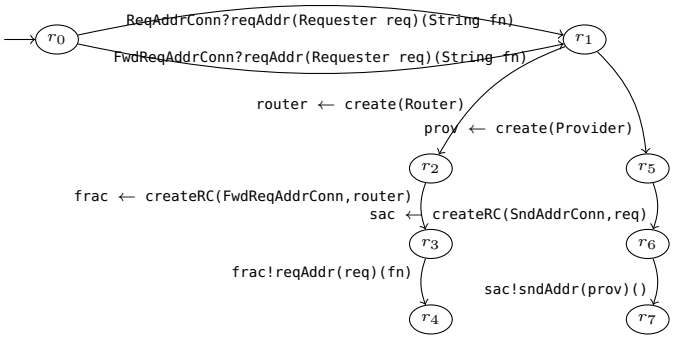
instance, Fig. 1c shows the behavior specification of a `Router`. Initially, a router is able to receive a request for an address either via the role connector `ReqAddrConn` (from the requester) or via `FwdReqAddrConn` (from another router). Depending on whether the router knows the peer storing `fn` or not, it either creates a provider role instance `prov` and sends it back to the requester (right branch) or it forwards the request to another router (left branch). The formal ensemble specification serves as an analysis model, e.g. to eliminate collaboration mismatches between different roles at early stages, and as a design model for implementation.



(a) Role type for a `Router`



(b) Ensemble structure for file retrieval



(c) Role behavior for a `Router`

Fig. 1: Ensemble specification in graphical notation (excerpt)

For the implementation and execution of ensembles, we provide the Java framework jHELENA [6]. The framework contains two layers and a system manager; cf. upper part of Fig. 2. The `metadata` layer implements the types used in ensemble structures, i.e. component types, role types, etc. All types and ensemble structures themselves are represented by objects of the `metadata` classes which are linked according to the formal definitions. While the `metadata` layer is related to the type level, the `developer interface` is related to the instance level. It contains abstract base classes which must be extended to implement subclasses for particular components, roles etc. The `SysManager` class provides basic functionality for the administration of ensembles. Its abstract operations must be implemented by a concrete system manager for configuring particular ensemble structures and the necessary types, creating the underlying component instances and instantiating and starting an ensemble. The framework controls that the created ensembles are built in accordance with previously configured ensemble structures.

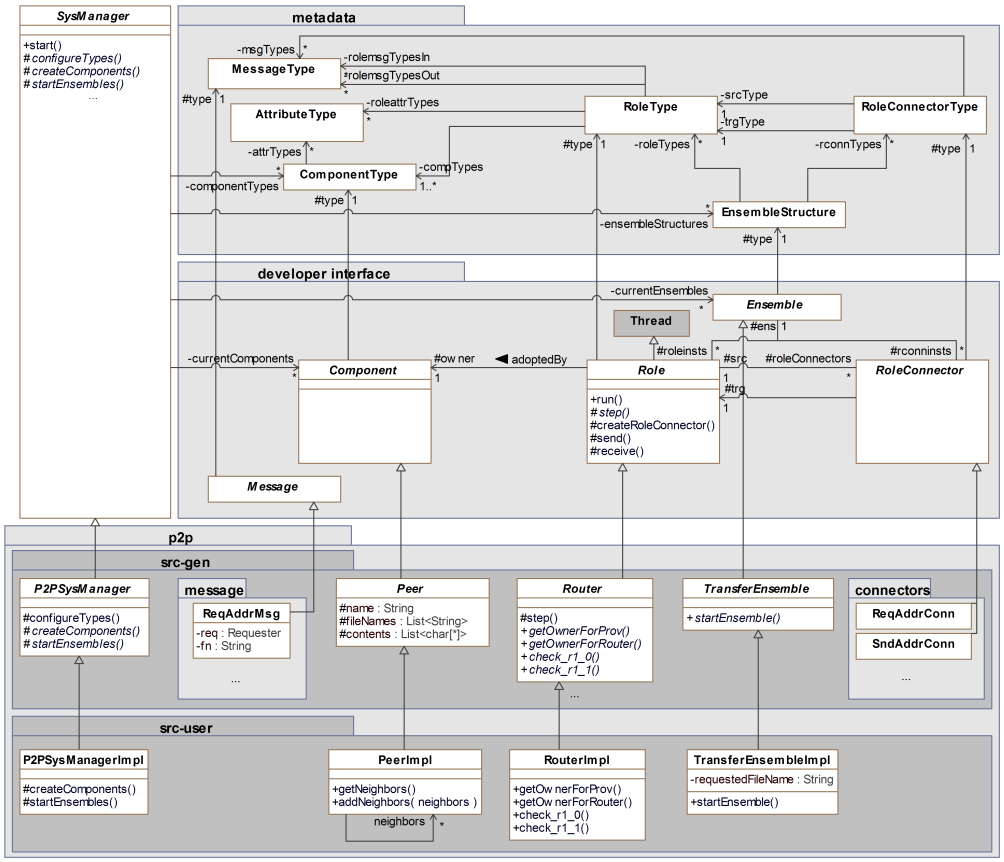


Fig. 2: jHELENA framework and generated p2p ensemble application (excerpts)

3 HELENATEXT and Code Generation

When modeling and implementing an ensemble-based system according to HELENA, the developer may experience two pitfalls. Without any editor support, the developer has to ensure herself that her specifications conform to HELENA and respect all constraints formulated in the formal definitions. To implement an ensemble, she has to translate an ensemble specification to jHELENA code by hand and has no guarantee that the implementation indeed respects the formal specification. We therefore define HELENATEXT, a domain-specific language (DSL) which provides a concrete syntax for ensemble specifications supporting roles and ensemble structures as first-class citizens. We also provide Eclipse integration which features a full HELENATEXT editor including syntax highlighting, content assist, and validation. Moreover, we define a set of rules for the automatic generation of jHELENA code from HELENATEXT.

HelenaText. For defining the syntax of HelenaText we use Xtext (`www.eclipse.org/Xtext/`), a framework for the development of DSLs fully integrated into Eclipse. We define a grammar in a BNF-like notation following the formal definitions of types, ensemble structures and role behaviors. Constraints which cannot be included into the DSL grammar are formulated as validation rules written with Xtend. For instance, List. 1.1 shows the grammar for the declaration of ensemble structures which must start with the key word `ensembleStructure` followed by its name. In curly braces the two parts (*roleTypes*, *rconnTypes*) of an ensemble structure Σ (cf. Sec. 2) are specified: `roleTypes` is a list of role types with multiplicity, `rconnTypes` is a list of role connector types (their specifications including source and target types are not shown). However, in the DSL grammar we cannot express the constraint that each role connector type must be equipped with a source and a target role type defined in *roleTypes*. For that, a validation rule in Xtend is added (cf. List. 1.2) which iterates over all role connector types in the ensemble structure and reports an error if the context condition is not satisfied. The concrete syntax for the declaration of the ensemble structure of Fig. 1b is shown in List. 1.3. The concrete syntax for role behaviors is a textual representation of labeled transition systems not shown here. The rules for all syntactic constructs of HelenaText can be found at [7].

```
1 EnsembleStructure:
2   'ensembleStructure' name=ValidID '{'
3     'roleTypes' '=' '{'
4       roleTypes+=RoleTypeWithMultiplicity (',' roleTypes+=RoleTypeWithMultiplicity)*'}';'
5     'rconnTypes' '=' '{'
6       rconnTypes+=[RoleConnectorType] (',' rconnTypes+=[RoleConnectorType])*'}';'
7   '}';
```

Listing 1.1: HelenaText grammar rule for ensemble structures

```
1 @Check
2 def check_es_rtsContainRcSrcAndTrgRoles(EnsembleStructure es) {
3   var rts = es.roleTypes.getRoleTypeList;
4   for (RoleConnectorType rct : es.rconnTypes) {
5     if (!(rts.contains(rct.srcType) && rts.contains(rct.trgType))) {
6       error('srcType and trgType of roleConnectorType not listed in roleTypes',...)
7 }}}
```

Listing 1.2: Validation rule for ensemble structures

```
1 ensembleStructure TransferEnsemble {
2   roleTypes = {<Requester,1>,<Router,1..*>,<Provider,0..1>};
3   rconnTypes = {ReqAddrConn,SndAddrConn,FwdReqAddrConn,ReqFileConn,SndFileConn};
4 }
```

Listing 1.3: Ensemble structure for file retrieval in HelenaText

Code generation. The code generator takes a HelenaText file containing a particular ensemble specification and generates a package for the ensemble application which is split into two parts, the (sub)packages `src-gen` and `src-user`; see Fig. 2. The package `src-gen` is already complete and must not be touched by the user. It contains a subclass (here `P2PSysManager`) of the `SysManager` class

which implements the method `configureTypes()`. The method body creates objects for the metadata classes to represent types and the ensemble structure in accordance with the specification. Moreover, `src-gen` contains subclasses for the abstract base classes of the developer interface. These subclasses, like `Peer`, `Router`, correspond to the types of the given ensemble structure.

To define templates for the code generation, we use XPAND. List. 1.4 shows an excerpt of such an XPAND rule. The operation `body` is called for any role type given in a HELENATEXT specification and generates the corresponding class declaration in jHELENA. Basically anything in the operation `body` is written to the generated class file except text enclosed in tag brackets «» which must be evaluated first. For example, in line 3 the class-header is built. The name of the class is dynamically evaluated from the expression «`classname`». This is a function of `RoleType` which is called for the first parameter `it` of the operation (see line 1) and retrieves the name of the role type `it` (the resulting class-header for the role type `Router` is shown in line 1 of List. 1.5). Afterwards, in line 4-6 of the XPAND rule all attributes of the role type are generated (which are none for the role type `Router`). Lines 8-18 declare additional attributes for any created instances or parameters of incoming messages in the role behavior of the role type such that their values can be accessed throughout the execution of the role behavior. For example, for the role behavior of the `Router` in Fig. 1c we need attributes to store the values of the two created role instances `router` and `prov`, of the role connector instances `frac` and `sac` as well as of the parameters `req` and `fn` of the incoming message `reqAddr`. For the role behavior itself the method `step` is generated from the textual labeled transition system representation in HELENATEXT (see line 22, template not shown here). Basically, a simple state machine is implemented which will be called repeatedly by the `run` method implemented in the base class `Role` of the developer interface in jHELENA.

```
1  def body(RoleType it, ImportManager im)
2  '''
3  public abstract class «it.classname» extends Role {
4    «FOR a:it.roleattrs»
5      «attrTypeGenerator.compile(a,im)»
6    «ENDFOR»
7
8    «IF it.roleBehavior != null»
9      «var instsAndParams = it.roleBehavior.getInstancesAndBindingParams(null, null)»
10     «FOR instOrParam : instsAndParams»
11       «IF(instOrParam instanceof AbstractInstance)»
12         «var inst = instOrParam as AbstractInstance»
13         «attrVisibility» «inst.type.name» «inst.name» = null;
14       «ELSEIF (instOrParam instanceof AbstractParam)»
15         ...
16       «ENDIF»
17     «ENDFOR»
18   «ENDIF»
19
20   public «it.classname»(Ensemble ens){ ... }
21
22   protected synchronized void step() throws ... { ... }
23 }'''
```

Listing 1.4: Generation rule for role types (excerpt)

```
1  public abstract class Router extends Role {
2    protected Router router = null;
3    protected FwdReqAddrConn frac = null;
4    protected Provider prov = null;
5    protected SndAddrConn sac = null;
6    protected Requester req = null;
7    protected String fn = null;
8
9    public Router(Ensemble ens) { ... }
10
11   protected synchronized void step() throws ... {
12     if (this.currentState == RouterState.r0) {
13       ReqAddrMessage reqAddr = (ReqAddrMessage) this.receive(
14           new ExpectedMsgTypes(ReqAddrConn.class, ReqAddrMessage.class),
15           new ExpectedMsgTypes(FwdReqAddrConn.class, ReqAddrMessage.class));
16       this.currentState = RouterState.r1;
17     }
18     else if (this.currentState == RouterState.r1) {
19       if (check_r1_0()) {
20         this.router = this.ens.createRole(RouterImpl.class, this.getOwnerForRouter());
21         this.currentState = RouterState.r2;
22       }
23       else if (check_r1_1()) {
24         this.prov = this.ens.createRole(ProviderImpl.class, this.getOwnerForProv());
25         this.currentState = RouterState.r5;
26   ...
```

Listing 1.5: Generated jHELENA code for a `Router` (excerpt)

Lines 11-26 in List. 1.5 show an excerpt of the `step` method generated from the behavior specification of `Router` shown graphically in Fig. 1c. The code generator creates a sequence of case distinctions to determine the next action depending on the current state. If there is only one transition starting from the current state, the action can directly be translated from HELENATEXT to code. If there are several alternatives for one state, like for `r0` or `r1` in Fig. 1c, the nondeterminism between those branches has to be resolved. In HELENA there are no mixed states in a role behavior meaning that whenever an incoming message is an alternative in a certain state then the other alternatives must also be incoming messages. Nondeterminism for incoming messages can be resolved easily by waiting for several messages in parallel; cf. line 13-15 in List. 1.5. For all other actions, the code generator cannot decide which transition to take. Therefore, for each such branch an abstract `boolean` method is called, cf. line 19 and 23, which must be implemented by the user to decide which branch should be taken. This mechanism is also used for the creation of new role instances. In fact, the user has to decide on which component the role instance should be deployed; cf. call to the abstract method `getOwnerForRouter()` in line 20. To implement user decisions, the code generator constructs the package `src-user` which includes implementation classes for all abstract classes in `src-gen`. The package `src-user` also contains a concrete manager class (here `P2pSysManagerImpl`). The user has to implement the methods `createComponents()` and `startEnsembles()` for creating components and for creating and starting ensembles, which can run concurrently. We have only described here the basic ideas behind the code generation. Formally it is based on a set of generation rules written in XPAND and XTEND which define, for each model element in HELENATEXT, how it is trans-

lated to jHELENA code. The rules for all syntactic constructs of HELENATEXT can be found at [7].

Next steps. In the near future we intend to provide a graphical DSL in addition to HELENATEXT which implements our UML-like notation used throughout the paper. Moreover, we want to investigate collaboration requirements and integrate tools for the analysis of ensemble specifications to check the absence of collaboration errors.

References

1. The ASCENS Project (2014), http://www.ascens-ist.eu
2. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: DEECo: an Ensemble-based Component System. In: Proceedings of 16th International Symposium on Component-Based Software Engineering. pp. 81–90. ACM (2013)
3. 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.) 10th International Symposium on Formal Methods for Components and Objects. Lecture Notes in Computer Science, vol. 7542, pp. 25–48. Springer (2011)
4. 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)
5. Hennicker, R., Klarl, A.: Foundations for Ensemble Modeling - The Helena Approach. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software. Lecture Notes in Computer Science, vol. 8373, pp. 359–381. Springer (2014)
6. Klarl, A., Hennicker, R.: Design and Implementation of Dynamically Evolving Ensembles with the Helena Framework. In: Proceedings of the 23rd Australasian Software Engineering Conference. pp. 15–24. IEEE (2014)
7. Klarl, A., Hennicker, R.: The Helena Framework (2014), http://www.pst.ifi.lmu.de/Personen/team/klarl/helena
8. Klarl, A., Mayer, P., Hennicker, R.: Helena@Work: Modeling the Science Cloud Platform. In: International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. Springer (to appear 2014)
9. 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.) 10th International Symposium on Formal Methods for Components and Objects. Lecture Notes in Computer Science, vol. 7542. Springer (2012)
10. Yoshida, N., Hu, R., Neykova, R., Ng, N.: The Scribble Protocol Language. In: Abadi, M., Lafuente, A.L. (eds.) 8th International Symposium Trustworthy Global Computing. Lecture Notes in Computer Science, vol. 8358, pp. 22–41. Springer (2013)