Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 00 (2015) 1–21
Sci. Comput. Program.

Implementation Framework and Code Generator for HELENA Ensemble Specifications

Annabelle Klarl, Rolf Hennicker, Lucia Cichella[1]

Ludwig-Maximilians-Universität München, Germany

Abstract

Ensembles are collections of autonomic entities which collaborate to perform certain tasks. They exhibit a complex dynamic behavior which is difficult to implement with a pure component-based approach. HELENA is a modeling approach for the specification of such ensembles. The conceptual key point of HELENA is that components can adopt different roles and can participate (under certain roles) in several, possibly concurrently running ensembles. In this paper, we present the jHELENA framework for the realization of ensemble specifications implementing roles as Java threads on top of a component. The framework contains a metadata layer, defining the structural aspects of an ensemble, and a developer interface providing abstract classes that can be extended for the realization of concrete ensemble applications. In the second part of this work, we focus on the automation of ensemble implementations and provide a tool to generate jHELENA code from a HELENA ensemble specification. Relying on the XTEXT workbench of Eclipse, we introduce the domain-specific language HELENATEXT for writing ensemble specifications and provide an Eclipse plug-in featuring an editor and a code generator. Our approach is demonstrated by a peer-2-peer file system network which is used as a running example throughout the paper.

Keywords: Ensemble-based systems, implementation framework, code generation

1. Introduction

The combination of autonomic computing and global interconnectedness in systems with large numbers of distributed nodes poses new challenges to software engineers. We no longer administrate and maintain systems by hand. Systems are rather composed of autonomic nodes with the ability to manage themselves. The nodes are able to perceive their environment and adapt their behavior accordingly. At the same time, they interact with other nodes in the system to collaborate in teams for some global goal. Those teams have to cope with heterogeneity of participants and dynamic composition.

In the HELENA approach [1], we advocate a formal model for the specification of such systems which is centered around the notion of *roles*. It provides concepts to describe systems where components team up in many concurrently running ensembles to perform global goal-oriented tasks. To participate in an ensemble, a component plays a certain role. This role adds role-specific behavior to the component and allows collaboration with (roles of) other components. By switching between roles, the component changes its currently executed behavior. By adopting several roles

[1]This work has been partially sponsored by the European Union under the FP7-project ASCENS, 257414.
Email addresses: klarl@pst.ifi.lmu.de (Annabelle Klarl), hennicker@pst.ifi.lmu.de (Rolf Hennicker)

in parallel, a component concurrently executes different behaviors and participates at the same time in different ensembles. The introduction of roles thereby allows to focus on the particular tasks which components fulfill in specific collaborations and to structure the implementation of ensemble-based systems. To make the Helena concepts usable for the realization of ensemble-based systems, we seek to provide tool support for writing ensemble specifications which conform to the formal foundations of the Helena approach as well as for implementing and executing these ensemble specifications.

Contributions. Our contribution is twofold: (1) We provide the implementation and execution framework jHelena which is realized in Java and transfers the concepts of roles and collaborations in ensembles to an object-oriented realization. In jHelena, 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. The framework implements the structural and dynamic rules enforced by the formal modeling concepts of ensemble specifications and their semantics. It provides an interface for the developer to realize concrete ensemble-based applications according to the Helena approach and allows to execute them. (2) To ease the implementation of ensemble specifications, we provide a domain-specific language HelenaText together with a code generator to jHelena. HelenaText allows to write ensemble specifications by supporting roles and ensembles as first-class citizens and enforces conformance of the specification to the formal Helena approach. HelenaText relies on the Xtext workbench and is therefore fully integrated into Eclipse providing an editor and a code generator. We implemented the code generator to translate HelenaText specifications to jHelena code. The generated code base must only be instantiated with concrete components to make the ensemble-based application directly executable.

Outline. The paper first reviews the Helena specification approach in Sec. 2 and sketches its application to a peer-2-peer file system example. Sec. 3 describes the implementation framework jHelena and shows how it is applied to the p2p example. Then, in Sec. 4, we introduce the domain-specific language HelenaText and the code generator from HelenaText ensemble specifications to jHelena code. We finish with some concluding remarks and discussion on related work in Sec. 5.

Relation to Our Previous Work. This paper is a significant extension of the conference paper [2]. We have added a detailed description of the jHelena framework, which first has been presented (in a more restrictive version) in [3]. Compared to [2], we have extended the description of the code generator to explain the workflow for the construction of the tool and the underlying concepts in more detail. The current presentation of the jHelena framework, the HelenaText DSL and the code generator subsumes several progressed features of the Helena modeling approach which have not been considered before. We enforce the connection between components and roles, support synchronous and asynchronous communication between roles, and specify role behaviors by process terms including guarded choice of branches. Due to these extensions, we can now generate the complete jHelena code for the implementation of role behaviors while, in [2], we had to leave open some implementation decisions for the user.

2. Ensemble Modeling with Helena

The role-based modeling approach Helena [1, 3] provides concepts to describe systems of (a large number of) components teaming up in possibly concurrently running ensembles to perform global goal-oriented tasks. To participate in an ensemble, a component plays a certain role. This role adds role-specific behavior to the component and allows collaboration with (roles of) other components playing other roles. By switching between roles, a component changes its currently executed behavior. By adopting several roles in parallel, a component concurrently executes different behaviors. In this section, we summarize the basic ideas and ingredients of the Helena approach.

Notation 1. *Whenever we work with tuples $t = (t_1, \ldots, t_n)$, we may use the notation $t_i(t)$ to refer to the value t_i of t.*

2.1. Ensemble Structures

Components form the basic layer of our approach. They provide basic capabilities to store data in attributes and associations to other components as well as to perform operations. Thus, component instances form the persistent foundation of ensemble-based systems. Components store data and associations which persist across the life-time of

different dynamically evolving ensembles and provide their computing resources in the form of operations (which can be exploited by the roles that components play in ensembles). To classify component instances we use component types. A *component type* is a tuple $ct = (ctnm, ctattrs, ctassocs, ctops)$ such that *ctnm* is the name of the component type, *ctattrs* a set of attribute types, *ctassocs* a set of associations to other component types, and *ctops* a set of operation types (consisting of an operation name, a formal parameter list, and possibly a return type).

For performing certain tasks, components team up in *ensembles*. Each participant in the ensemble contributes specific functionalities to the collaboration, we say, the participant plays a certain *role* in the ensemble. Roles are classified by their type. A role type determines the types of the components that are able to adopt this role. It also defines role-specific attributes (to store data that is only relevant for performing the role), and message types for incoming and outgoing messages supported by this role type. Thereby, a *message type* is of the form $msg = msgnm(riparams)(dataparams)$ such that *msgnm* is the name of the message type, *riparams* is a list of (typed) formal parameters to pass role instances, and *dataparams* is a list of formal parameters to pass ordinary data. Formally, a *role type rt* over a given set of component types CT is a tuple $rt = (rtnm, rtcomptypes, rtattrs, rtmsgs)$ such that *rtnm* is the name of the role type, $rtcomptypes \subseteq CT$ is a finite, non-empty subset of component types (whose instances can adopt the role), *rtattrs* is a set of role-specific attribute types, and $rtmsgs = (rtmsgs_{out}, rtmsgs_{in})$ specifies two sets of message types, one for incoming and one for outgoing messages supported by the role type *rt*.

To define the structural characteristics of collaborations, an *ensemble structure* determines the type of an ensemble that is needed to perform a certain task. It specifies which role types are needed in the collaboration and how many instances of each role type may (or must) contribute (given by a multiplicity from UML). The roles contributing to the ensemble can then exchange messages which are outgoing at the source role and incoming at the target role. Interacting role instances can use synchronous or asynchronous communication via input queues. An ensemble structure specifies, for each role type, the (finite) capacity of the input queue of each role instance of that type (the value 0 expresses synchronous communication). An ensemble structure is always built on top of a given set CT of component types whose instances can adopt roles as specified in the ensemble structure.

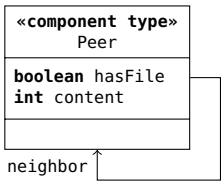
Definition 1 (Ensemble Structure). *Let CT be a set of component types. An* ensemble structure Σ over CT *is a tuple* $\Sigma = (nm, roletypes, roleconstraints)$ *such that nm is the name of the ensemble structure, roletypes is a set of role types over CT and for each* $rt \in roletypes$, $roleconstraints(rt)$ *is a pair of a multiplicity (from UML), like* 0..1, 1, $*$, 1..$*$, *etc. and the finite capacity* $c \in \mathbb{N}$ *of the input queue of rt.*

P2P Example. Let us illustrate the HELENA concepts at the example of a peer-2-peer network supporting the distributed storage of files which can be retrieved upon request. Several peers are connected in a ring structure and 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 component instances of the type `Peer`. Fig. 1 shows the component type `Peer` and all three role types in a UML-like graphical notation. For simplicity, we only consider peers which can store one single file. The attribute `hasFile` of a `Peer` (cf. Fig. 1a) indicates whether the peer has the file independently from the file's content information represented by the attribute `content`. A `Peer` is furthermore connected to its neighbor depicted by the association `neighbor`. The three role types `Requester`, `Router`, and `Provider` indicate by the notation `RoleType:{Peer}` that any component instance of type `Peer` can adopt the role `RoleType`. The `Requester`, for instance, stores in its attribute `hasFile` whether it already has the file and it supports two incoming and two outgoing messages which will be explained below.

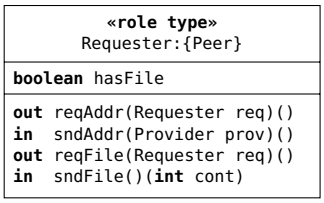
Fig. 2 shows a graphical representation of the ensemble structure $\Sigma_{transfer}$ for the p2p example. It consists of the three role types `Requester`, `Router`, and `Provider` where we now associate multiplicities and capacities for their input queues. For instance, exactly one `Requester` is required in a file transfer ensemble while arbitrarily many `Router`s might be necessary to route messages through the network. The input queue of a `Requester` or `Router` can store up to two messages, of a `Provider` only one message. Arrows denote the messages which can be exchanged between the roles. For instance, the `Requester` can send the message `reqAddr(Requester req)()` to a `Router`. This message will be used for requesting the address of a provider for the requested file such that the file can be directly downloaded afterwards using the messages between `Requester` and `Provider`.

2.2. Ensemble Specifications

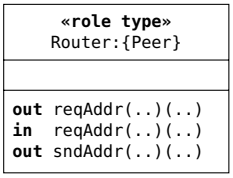
An ensemble specification adds dynamic behavior to an ensemble structure by equipping each role type with a role behavior. A *role behavior* is given by a process expression built from the null process, action prefix, guarded



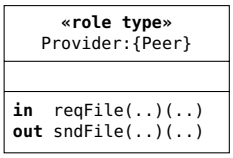
(a) Component type `Peer`



(b) Role type `Requester`



(c) Role type `Router`



(d) Role type `Provider`

Figure 1: All types for the p2p example

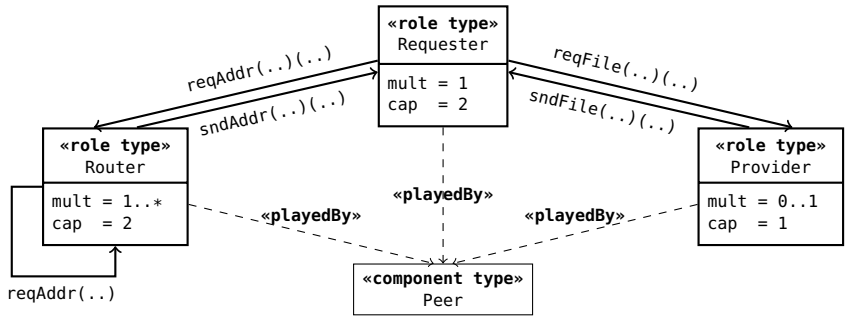


Figure 2: Ensemble structure $\Sigma_{transfer}$ for the p2p example

choice (branch is nondeterministically selected if several branches are executable), and process invocation. Guards are predicates over component or role attributes. There are seven different kinds of actions: creation and retrieval of role instances, sending (!) and receiving (?) a message, operation calls on the owning component, and setting attribute values for the role instance itself or the owning component. These actions must fit to the declared ensemble structure, e.g., messages can only be sent by roles which declare them.[2] We additionally use predefined expressions like **self** to refer to the current role instance and **owner** to refer to the owning component instance. The attributes of the current role instance and its owning component instance are accessed in a Java like style and we provide a predefined query **plays**(rt, ci) to request whether the component instance *ci* currently plays the role *rt*.

Definition 2 (Role Behavior). *Let* Σ *be an ensemble structure and rt be a role type in* Σ*. A* role behavior $RoleBeh_{rt}$ *for rt is a process expression built from the following abstract syntax:*

$P ::= \textbf{\textit{nil}}$	*(null process)*
$\mid a \,.\, P$	*(action prefix)*
$\mid$ ***if*** (*condition1*) ***then*** $\{P1\}$ (***or*** (*condition2*) ***then*** $\{P2\})^*$	*(guarded choice)*
$\mid N$	*(process invocation)*

[2] In the following, we use X, Y for role instance variables, rt' for role types, x for data variables, E, F for role instance expressions, e for data expressions, and *ci* for component instances (assuming a given repository of those); $\vec{z}$ denotes a list of z.

$$
\begin{array}{lll}
a ::= & | \; X{\leftarrow}\textbf{\textit{create}}(rt', ci) & \textit{(role instance creation)} \\
& | \; X{\leftarrow}\textbf{\textit{get}}(rt', ci) & \textit{(role instance retrieval)} \\
& | \; F!msgnm(\vec{E})(\vec{e}) & \textit{(sending a message)} \\
& | \; ?msgnm(\vec{rt'}\vec{X})(\vec{x}) & \textit{(receiving a message)} \\
& | \; [x =] \; \textbf{\textit{owner}}.opnm(\vec{e}) & \textit{(component operation call)} \\
& | \; \textbf{\textit{owner}}.attr = e & \textit{(component attribute setter)} \\
& | \; \textbf{\textit{self}}.attr = e & \textit{(role attribute setter)}
\end{array}
$$

The full ensemble specification in Helena consists of two parts: an ensemble structure describing the structural composition of the ensemble and a set of role behavior declarations describing the interaction behavior of the ensemble by introducing a dynamic behavior for each role type occurring in the ensemble structure.

Definition 3 (Ensemble specification). *An* ensemble specification *is a pair* $EnsSpec = (\Sigma, RoleBeh)$ *such that* Σ *is an ensemble structure and a family* $RoleBeh$ *of* role behavior specifications $RoleBeh_{rt}$*, one for each role type* rt *of* Σ*.*

P2P Example. The full ensemble specification for the p2p example can be found on our website [4]. Exemplarily, we here discuss the behavior specification of a `Router` (cf. Fig. 3). Initially, a router is able to receive a request for an address of the provider of the requested file. Depending on whether the router's owner has the file or not, it either provides the file to the requester in the process $P_{provide}$ or forwards the message to another router P_{fwd}. To provide the file in $P_{provide}$, the router creates a new role instance of type `Provider` on its owning component and sends the reference of the newly created provider back to the requester. To forward the message in P_{fwd}, the router checks whether the neighboring component of its owner already plays the role `Router`. If so, the neighboring component does not have the file (since it already forwarded the message in its role as a router) and the router can stop to forward the message (represented by **nil**). That means for the whole ensemble that the file does not exist in the p2p network. If the neighboring component does not play the role `Router`, a new router is created on the owner's neighbor and the request is forwarded to this new router (cf. process P_{create}). Afterwards, it resumes its behavior from the beginning.

$$
\begin{aligned}
RoleBeh_{router} &= ?reqAddr(Requester\ req)() \ . \\
&\quad \textbf{if } (\textbf{owner}.hasFile) \textbf{ then } \{P_{provide}\} \\
&\quad \textbf{or } (!\textbf{owner}.hasFile) \textbf{ then } \{P_{fwd}\} \\
\\
P_{provide} &= prov{\leftarrow}\textbf{create}(Provider, \textbf{owner}) \ . \\
&\quad req!sndAddr(prov)() \ . \ \textbf{nil} \\
\\
P_{fwd} &= \textbf{if } (\textbf{plays}(Router, \textbf{owner}.neighbor)) \textbf{ then } \{ \ \textbf{nil} \ \} \\
&\quad \textbf{or } (!\textbf{plays}(Router, \textbf{owner}.neighbor)) \textbf{ then } \{ \ P_{create} \ \} \\
\\
P_{create} &= router{\leftarrow}\textbf{create}(Router, \textbf{owner}.neighbor) \ . \\
&\quad router!reqAddr(req)() \ . \ RoleBeh_{router}
\end{aligned}
$$

Figure 3: Role behavior for the router in the p2p example

2.3. Ensemble Automata

Helena ensemble specifications are semantically interpreted by ensemble automata [1]. Ensemble automata are labeled transition systems describing the evolution of ensemble states. Intuitively, given a set of component instances, the states of an ensemble automaton capture the currently existing role instances of each role type occurring in the ensemble, for each role instance, a unique component instance which currently adopts this role, the data values

currently stored for each attribute of the role instance, the local environment for local variables of this role instance, the current content of the input queue of the role instance, and the current progress of execution according to the specified role behavior for each role instance. Transitions between ensemble states are triggered by role instance creation or retrieval, communication actions, operation calls, and (role or component) attribute update. For message exchange, we assume binary communication between role instances, but support both, synchronous (rendezvous) and asynchronous, communication. For that purpose, each instance *ri* of a role type *rt* is equipped with an input queue to which messages addressed to *ri* are delivered. In the synchronous case, the sender is blocked until the receiver has consumed the message. However, it is important to note that the communication style is not determined by an ensemble specification since the role behaviors specify local behaviors.

Ensemble States. Let us now look more closely to the formal definition of ensemble states. For this purpose, we assume given an ensemble structure Σ and a set $INST$ of component instances. Since component instances are of global nature and can be used across different ensembles, component instances and their local states do not belong themselves to an ensemble state, but are used as a basic layer.

An ensemble state σ (over Σ and $INST$) describes the local states of all currently existing role instances. A local state of a role instance *ri* is a tuple $(rt, ci, roledata, \rho, q, ctrl)$ which stores the following information: (1) The role type *rt* of the instance, (2) the component instance $ci \in INST$, which currently adopts the role, (3) the current data values *roledata* stored by *ri* (for the attributes *rtattrs* of *rt*), (4) the local environment ρ of *ri* mapping local variables to values, (5) the current content q of the input queue of the role instance *ri*, and (6) the current control state *ctrl* showing the next step in the role behavior execution of *ri*. We denote by Λ the set of possible local states.

We assume that a role instance is represented by a unique identifier which stems from a universal, countably infinite set *RID*. Then, an *ensemble state* representing the local states of all currently existing role instances is given by a finite function $\sigma : RID \to \Lambda$. Finiteness of σ means that σ is defined only for a finite subset $roleinsts(\sigma) \subset RID$ which represents the currently existing role instances of σ. We say that a component instance $ci \in INST$ participates in an ensemble in state σ if *ci* occurs in the local state of a role instance $ri \in roleinsts(\sigma)$.

P2P Example. We illustrate the definition of an ensemble state at our p2p network. We assume given four component instances of type `Peer`, i.e. $INST = \{\texttt{p1}, \texttt{p2}, \texttt{p3}, \texttt{p4}\}$. Then, an ensemble state could be that `p1` has adopted the role of a requester (in terms of the role instance `req`); `p2` and `p3` work as routers, adopting the role instances `rout1` and `rout2` resp., and `p3` provides the file and has, additionally to its router role, currently adopted the role instance `prov`; component `p4` is not involved in this collaboration. A graphical representation of this state (not showing local environments and input queues) is depicted in Fig. 4. The current control state of each role instance is shown in an ellipse. We omit the representation of the local environments and the current content of the input queues.

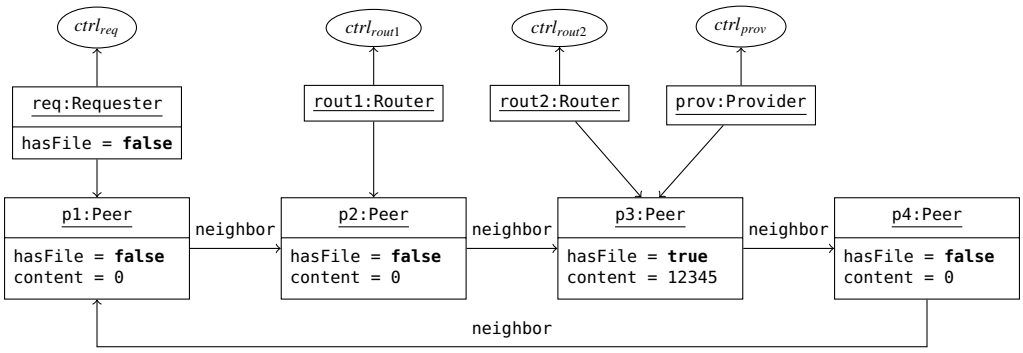


Figure 4: An ensemble state over {p1,p2,p3,p4} for the ensemble structure $\Sigma_{transfer}$ in graphical notation

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 Sec. 3. The states of an ensemble automaton are ensemble states as defined above. Transitions between ensemble states are

triggered by ensemble actions which are either (a) management actions, for creating and retrieving role instances, (b) communication actions, for sending and receiving messages of collaborating role instances, (c) operation calls from role instances to their owning components, or (d) setting the values of (component and role) attributes.

(a) Ensemble automata handle role instance creation and retrieval as expected by introducing a fresh role instance or retrieving an existing one. For instance, a create action is formally represented by a label of the form $ri{:}X{\leftarrow}\textbf{create}(rt', ci)$ expressing that the role instance *ri* has created a role instance of type rt' which is adopted by the component instance *ci*. The reference to the newly created role instance was bound to the variable *X*. (b) Communication actions realize binary communication between role instances. A send action is of the form $ri{:}F!msgnm(\vec{E})(\vec{e})$ such that *ri* is the sending role instance, *F* denotes the target role instance, $\vec{E}$ is a list of actual role instance parameters and $\vec{e}$ is a list of actual data parameters that has to match to the formal parameters of the message type referred to by *msgnm*. The effect of a send action is that the message is put in the input queue of the role instance referred to by *F*. It depends on the assumed communication style whether the sender is blocked until the receiver consumes the message (synchronous communication) or not (asynchronous communication). Similarly, a receive action is of the form $ri{:}?msgnm(\vec{rt'}\vec{X})(\vec{x})$ expressing that the role instance *ri* consumes the message from its input queue and binds the received parameters to $\vec{X}$ and resp. $\vec{x}$. (c) Operation calls have the form $ri{:}\textbf{owner}.opnm(\vec{e})$ such that **owner** is the owning component instance of *ri* and $\vec{e}$ is a list of actual data parameters for the call to the operation with name *opnm*. An operation call initiates an internal computation on a component instance and can only be issued from a role instance *ri* which is currently owned by the component instance. However, component instances may adopt several role instances (even from different ensembles) in parallel. Therefore, operation calls must be mutually exclusive over all role instances the component adopts. (d) Lastly, setting an attribute is described by $ri{:}\textbf{owner}.attr = e$ resp. $ri{:}\textbf{self}.attr = e$. Its effect is that the value of the corresponding attribute is set to the given value.

To evolve an ensemble automaton according to an ensemble specification, we start from a given initial state. This normally consists of a single role instance initiating the collaboration of the ensemble which then evolves in accordance to the behavior specifications of each of its role types.

3. Ensemble Implementation with the jHELENA Framework

jHELENA is a framework which allows to implement ensemble specifications on an object-oriented platform. The framework is written in Java and realizes the syntax and semantics of the HELENA modeling approach described in Sec. 2. The overall architecture of the framework is shown in Fig. 5. It contains two layers, the `metadata` layer and the `developer-interface` which both are used by a system manager.

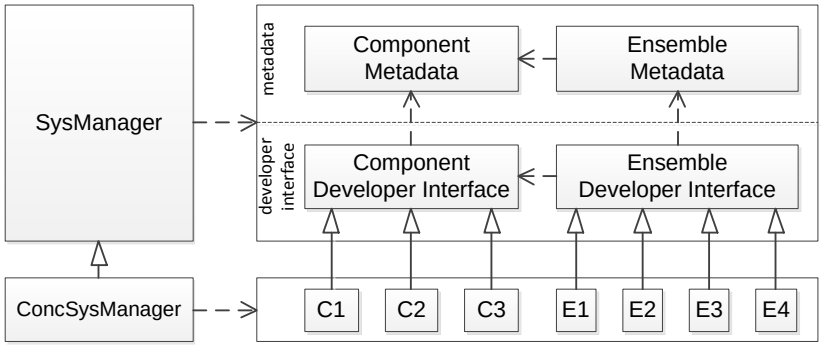


Figure 5: Architecture of the jHELENA framework

The `metadata` layer allows to define the meta model of ensemble specifications in terms of component types and ensemble structures (and thus role types etc.) according to the definitions in Sec. 2.1. The classes of this layer are used to describe the structural aspects of an ensemble structure while the framework guarantees that all conditions of the definitions are respected.

The `developer-interface` provides the basic functionality to realize an actual ensemble-based application. By extending the abstract base classes of the `developer-interface`, the developer implements concrete components, indicated by `C1,C2,C3`, as well as concrete ensembles, indicated by `E1,E2,E3,E4`. This layer also allows to realize role behaviors in terms of Java threads, following an ensemble specification in the sense of Sec. 2.2, and forces implementations to follow the abstract principles and conditions established by an ensemble automaton (cf. Sec. 2.3).[3]

The system manager and its concrete, application-dependent extension are responsible for the configuration of the component-based platform and ensemble structures, the creation of initial ensemble states, and the launch of concrete ensemble-based applications running ensembles concurrently on top of the component-based platform. Extending the abstract `SysManager` guides the user through the development process: the user first needs to implement the method `configureTypes()` (cf. Fig. 6) which configures all structural types for the application; afterwards the method `createComponents()` initializes all components providing the component-based platform for the application-specific ensembles; lastly, the initial state of each ensemble is established and the ensembles themselves are launched in the method `startEnsembles()`. With this method, many concurrently running ensembles can be started one after the other.

In this section, we first discuss how the formal definitions of ensemble structures are realized in the `metadata` layer of jHELENA. Afterwards, we explain the infrastructure that the `developer-interface` provides for the implementation of actual ensemble-based applications. Lastly, we exemplify the usage of jHELENA at our running p2p example.

3.1. Metadata Layer

The upper package of Fig. 6 gives an overview of the `metadata` layer. All types used to build ensemble structures are realized by corresponding `metadata` classes; the relationships between types are represented by associations in the `metadata` layer of the framework. Hence, this layer defines the meta model of ensemble structures. Concrete instances of classes on this layer represent the types contributing to the ensemble-based system (and not the actual instances of the types).

For example, to represent role types $rt = (rtnm, rtcomptypes, rtattrs, rtmsgs)$ we use the class `RoleType`. The name *rtnm* 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` (cf. Sec. 3.2). The set *rtcomptypes* 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 *rtattrs* are determined by the association with end `attrTypes` directed from `RoleType` to the class `AttributeType`. Similarly, the sets of message types $rtmsgs_{out}$ and $rtmsgs_{in}$ occurring in *rtmsgs* are modeled as associations with end `msgTypesOut` and `msgTypesIn` 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 the static factory method `createType` of the class `RoleType` (not shown in the diagram) such that the actual parameters point to objects representing the constituent parts of a role type.

An ensemble structure $\Sigma = (nm, roletypes, roleconstraints)$ is represented by an object of the corresponding class `EnsembleStructure`. It has an association with end `roleTypes` to navigate to the role types needed to contribute to the ensemble structure.

Similarly, all types of an ensemble-based system are realized in the `metadata` layer.

[3] 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 from left to right.

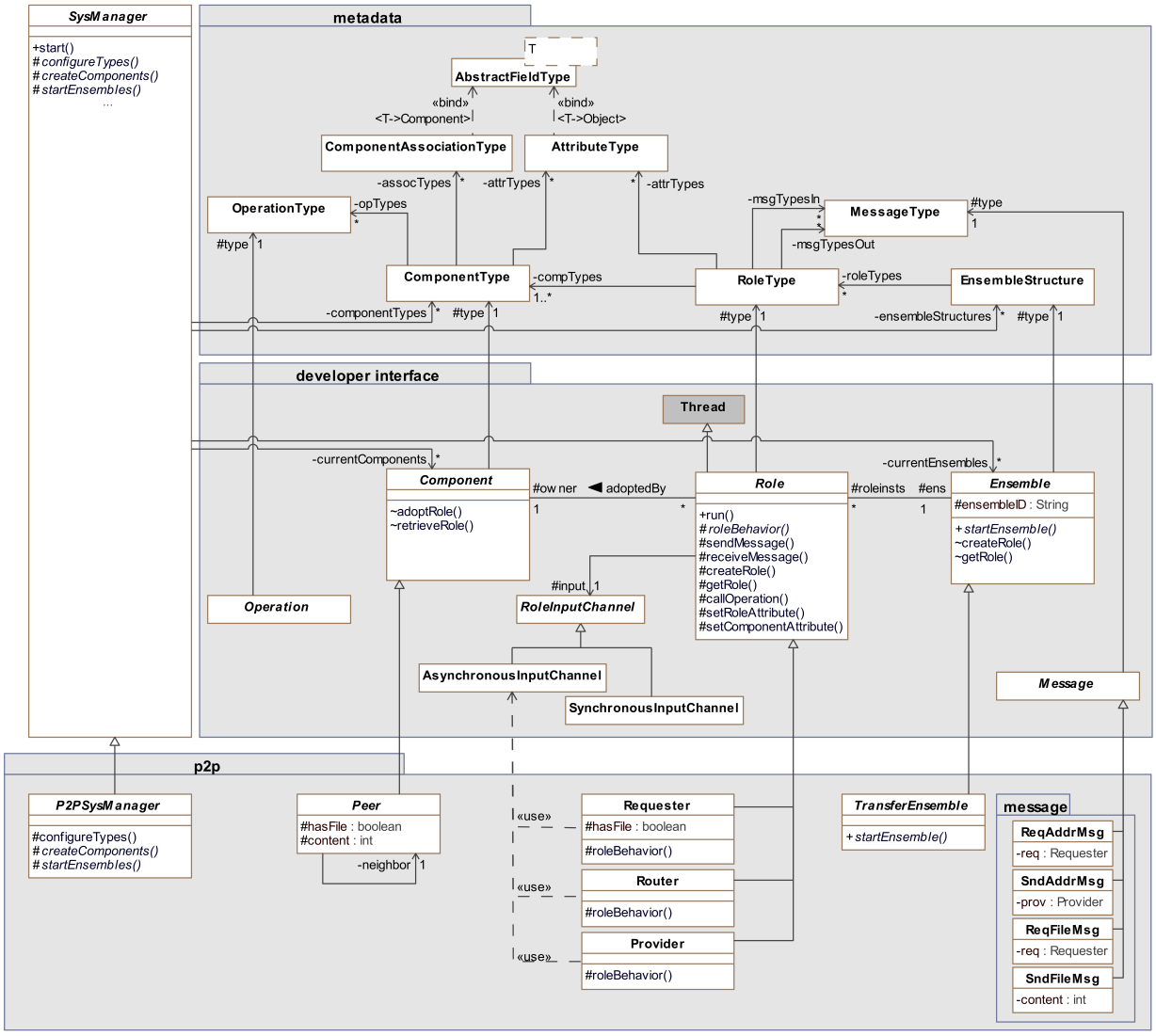


Figure 6: jHELENA framework and its application to the p2p example

3.2. Developer Interface

The goal of the `developer-interface` layer 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 are extended to represent concrete components, ensembles, roles and messages, i.e., instances of those classes are actual instances of components etc. An overview of the `developer-interface` is shown in the second package of Fig. 6. It offers abstract classes `Component`, `Ensemble`, `Role`, etc. for each `metadata` class, apart from `AttributeType`[4]. Each abstract class has an association to the corresponding `metadata` class representing the `type`

[4]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. Operations, however, are represented by the abstract class `Operation` although they are represented by Java methods associated to component instances. This is mainly because calls to operations as actions in a role behavior can then explicitly be validated by the framework. They are issued by passing an instance of the class `Operation` and call the corresponding method of the component by the reflection mechanism of Java.

of an instance. To implement an actual ensemble-based application, the abstract classes of the `developer-interface` must be extended by concrete subclasses as indicated by the inheritance arrows in Fig. 6. The framework ensures, using the reflection mechanism of Java, 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 jHELENA is guided by the idea that any running ensemble can be abstractly considered as an ensemble automaton. Hence, the states of a concrete ensemble must reflect the formal definition of an ensemble state in Sec. 2.3 and the execution steps have to correspond to the transitions of an ensemble automaton.

Ensemble States. Let us first consider how ensemble states are reflected in jHELENA. Formally, an ensemble state is a finite function $\sigma : RID \to \Lambda$. In jHELENA, an ensemble is represented by an instance of the class `Ensemble`. The set of role instances $roleinsts(\sigma)$ currently existing in the ensemble is given by the association with end `roleinsts` directed from `Ensemble` to the class `Role`. Formally, the local state of a role instance in $roleinsts(\sigma)$ is given by a tuple $(rt, ci, roledata, \rho, q, ctrl)$ as explained in Sec. 2.3. In jHELENA, the role type rt of a role instance can by accessed via the association from the class `Role` to the class `RolyType` with association end `type`. The association `adoptedBy` with end `owner` navigates to the unique component instance ci which currently adopts this role. The attribute values $roledata$ of a role instance are given by the current values of the instance variables of the concrete role class. The local environment ρ is implicitly given by another set of instance variables of the concrete role class which are accessed when the role behavior is executed. Similarly, the current control state $ctrl$ of each role instance is implicitly given by the program counter of the thread executing the role behavior. Finally, the input queue q is represented by the association with end `input` from `Role` to `RoleInputChannel`.

Ensemble Automaton. To advance an ensemble, each role executes its role behavior. For the realization of a role behavior the class `Role` prescribes the implementation of its abstract method `roleBehavior()`. All actions executed in a role behavior are reflected by calls to their method counterparts in the class `Role`. The corresponding methods implement the effect of those actions as described in Sec. 2.3. Furthermore, 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).

The execution steps of all roles of an ensemble together have to correspond to the transitions of an ensemble automaton. According to the definition of ensemble automata, four types of actions thereby advance the system: (a) management actions, (b) communication actions, (c) operation calls to the owning component, and (d) setting the values of (component and role) attributes.

(a) Management labels, like $ri{:}X{\leftarrow}\mathbf{create}(rt', ci)$, are implemented in jHELENA by calls to the corresponding method in the class `Role` (parameters are not shown). In this case, the `createRole()` method in the class `Role` issues a call to the `createRole()` method of the class `Ensemble`. The ensemble creates a new role instance, advises the component instance ci to adopt the new role instance, and starts the thread of the role instance representing the role behavior. The ensemble takes also care to validate the correctness of the action execution, e.g., that the component instance ci is allowed as an owner for the desired role type rt and that the number of currently existing role instances does not exceed the multiplicity specified in the ensemble structure. (b) Communication labels are of the form $ri{:}F!msgnm(\vec{E})(\vec{e})$ for sending a message and $ri{:}?msgnm(\vec{rt'X})(\vec{x})$ for receiving a message. They are represented by calls to the corresponding methods in the class `Role`. For sending a message, the target role and the message has to be given. While the role is represented by a reference to an existing role instance, the message part $msgnm(\vec{E})(\vec{e})$ is represented in jHELENA by an instance of (a subclass of) `Message` where message parameters are implemented as attributes. Internally, the action is first validated for well-formedness, e.g., that the sending role supports the message as an outgoing message. Afterwards, it is transmitted to the input queue of the receiving role. Depending on the used implementation for the input queue (`SynchronousInputChannel` or `AsynchronousInputChannel`), the sender is blocked until the message is consumed by the receiver. To receive the message, the receiver calls the corresponding method of the class `Role`. The expected type of the message has to be given. Internally, the role retrieves a waiting message from its input queue that matches the expected message type. (c) Operation calls are implemented in jHELENA by the method `callOperation()` of the class `Role`. The operation itself is represented by an instance of (a subclass of) `Operation` where operation parameters are implemented as attributes. Internally, the action is again first validated, e.g., whether the actual parameters match the formal parameters. Afterwards, the corresponding method on the owning component is called via the

reflection mechanism of Java. Note that we have to use reflection since operation calls are implemented on the framework level where the actual application-specific operations are not yet known. (d) Similarly, setting a (component or role) attribute is implemented by the corresponding method in the class `Role`. Internally, Java's reflection mechanism is used again to access the (component's or role's) attribute for the same reasons as before.

System Manager. Lastly, the abstract `SysManager` class provides the means to actually start an ensemble by the template method `start()`. This method sequentially calls the methods `configureTypes()` (to configure all structural types for the application), `createComponents()` (to initialize all components providing the underlying component-based platform) and `startEnsembles()` (to initialize and start concrete ensembles on top of the component-based platform). They all need to be implemented in a subclass of the `SysManager` according to the specific application.

3.3. Framework Application

We illustrate the use of the framework by implementing the p2p file transfer ensemble. We perform the implementation in two major steps concerning the structural aspects and the dynamic behavior.

Structural Aspects. For the static aspects, we first extend the classes of the `developer-interface` for each type in the example as shown in the package `p2p` in Fig. 6: `Peer` extends `Component`, `Requester`, `Router`, and `Provider` extend `Role`, etc. We define attributes as instance variables of component and role classes, (operations as methods of component classes,) and parameters of messages as attributes of the particular message classes. However, we do not realize the role behaviors yet.

Afterwards, we extend the abstract class `SysManager` by the class `PeerSysManager` and implement the method `configureTypes()` to configure all types of the p2p example. This method instantiates all type classes of the `metadata` layer and connects them appropriately to represent the ensemble structure $\Sigma_{transfer}$ in Fig. 2. An excerpt of the implementation is shown in Fig. 7. The method first has to create all component types underlying the ensemble-based system. For the p2p example, we instantiate only one component type for peers (instantiation of attribute types and component association types is shown inline as well as the empty set for operation types) and add it to the list `componentTypes` of the system manager by calling the method `addComponentType()` (cf. line 2-9 in Fig. 7). Afterwards, we create instances for all types of the ensemble structure and connect them accordingly. Line 11-21 in Fig. 7 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 24-25).

Dynamic Behavior. The second step is to add dynamic behavior. For this purpose, we realize the ensemble specification by first implementing the methods `roleBehavior()` of all concrete role classes and the methods representing operations of components. Afterwards, we indicate how to concretely start an ensemble by implementing `startEnsemble()` of the class `TransferEnsemble`. Lastly, we realize a concrete application by implementing the methods `createComponents()` and `startEnsembles()` of the class `P2PSysManager`.

Implementing the method `roleBehavior()` for each concrete role class essentially means deriving an appropriate branching sequence of methods calls representing actions from the process expressions used in the role behavior descriptions of the ensemble specification. Particularly interesting is the translation of guarded choice. In HELENA, all guards are evaluated and the executed branch is nondeterministically selected from all branches guarded by a condition evaluating to `true`. In jHELENA, nondeterminism cannot be reflected, thus the first branch guarded by a condition evaluating to `true` is selected. The method `roleBehavior()` in Fig. 8 shows the implementation of the behavior $RoleBeh_{router}$ of a router which was specified in Fig. 3. All guards are mutually exclusive, thus we do not suffer from sequentializing nondeterminism as explained before. Each action is translated to a call to its corresponding method in the class `Role`. Thereby, binding a value (e.g., from receiving an incoming message) is realized by setting a particular role instance variable. For example, the router receives the reference to the requester with the incoming message `reqAddrMsg` in line 11. To be able to forward this reference later on (e.g., in line 37), it is stored in the instance variable `req` in line 12. Lastly, to access (component and role) attributes, particular getters are called like the method `getNeighborOfOwner()` for retrieving the value of the attribute `neighbor` of the owner in line 27.

```
1  public void configureTypes() {
2    ComponentType peer = ComponentType.createType(Peer.class,
3         getAsSet(
4            AttributeType.createType("hasFile", Boolean.class),
5            AttributeType.createType("content", Integer.class)),
6         getAsSet(
7             ComponentAssociationType.createType("neighbor", Peer.class),
8         new HashSet<OperationType>());
9    this.addComponentType(peer);
10
11   Set<ComponentType> reqCompTypes = getAsSet(peer);
12   Set<AttributeType> reqAttrTypes = getAsSet(AttributeType.createType("hasFile", Boolean.class));
13   Set<MessageType> reqMsgsOut =
14        getAsSet(
15           MessageType.createType(ReqAddrMessage.class),
16           MessageType.createType(ReqFileMessage.class));
17   Set<MessageType> reqMsgsIn =
18        getAsSet(
19           MessageType.createType(SndAddrMessage.class),
20           MessageType.createType(SndFileMessage.class));
21   RoleType req = RoleType.createType(Requester.class, reqCompTypes, reqAttrTypes, reqMsgsOut,reqMsgsIn);
22   ...
23
24   EnsembleStructure transferEnsemble = EnsembleStructure.createType(TransferEnsemble.class, ...);
25   this.addEnsembleStructure(transferEnsemble);
26 }
```

Figure 7: Instantiation of types in the method `configureTypes()` of the class `P2PSysManager`

Operations of components are implemented as methods of the corresponding (subclass of the) class `Component`. They have to take the parameters of the operation as input. The body of the method implements the behavior of the operation which was not yet part of the ensemble specification, but has now to be added by the developer. In our p2p example, we do not have any operations.

The method `startEnsemble()` of the class `TransferEnsemble` actually starts an instance of the ensemble (cf. Fig. 9). 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.

Lastly, a concrete scenario needs to be set up. The system is populated by concrete peers in the method `createComponents()` of the `P2PSysManager` (cf. Fig. 10). Each peer is initialized as indicated in line 2-3, the network of peers as a ring structure is set up (line 4-5), and each peer is added to the list `currentComponents` (line 6-7). Afterwards, concrete ensemble instances are created and run in the method `startEnsembles()` (cf. Fig. 11).

Using our framework, the implementation of the p2p example was straightforward and could easily be derived from the formalization in HELENA. Different file transfer ensembles could be instantiated (cf. line 6 in Fig. 11) and run concurrently.

```
1  public class Router extends Role {
2    protected Requester req;
3    protected Provider prov;
4    protected Router router;
5
6    public Router(Ensemble ens, Integer capacity) throws ... {
7      super(ens, capacity);
8    }
9
10   protected void roleBehavior() throws ... {
11     ReqAddrMessage reqAddrMsg = (ReqAddrMessage) this.receiveMessage(ReqAddrMessage.class);
12     this.req = reqAddrMsg.getReq();
13
14     if (this.getHasFileOfOwner()) {
15       this.provide();
16     }
17     else if (!(this.getHasFileOfOwner())) {
18       this.fwd();
19     }
20   }
21   private void provide() {
22     this.prov = this.createRole(Provider.class, this.owner);
23
24     this.sendMessage(this.req, new SndAddrMessage(this.prov));
25   }
26   private void fwd() {
27     if (this.playsRole(Router.class, this.getNeighborOfOwner())) {
28       // do nothing
29    }
30     else if (!(this.playsRole(Router.class, this.getNeighborOfOwner()))) {
31       this.create();
32     }
33   }
34   private void create() {
35     this.router = this.createRole(Router.class, this.getNeighborOfOwner());
36
37     this.sendMessage(this.router, new ReqAddrMessage(this.req));
38
39     this.roleBehavior();
40   }
41 ...
42 }
```

Figure 8: Implementation of $RoleBeh_{router}$ in Fig. 3

```
1 protected void startEnsemble(Component initComp) {
2   Requester req = this.createRole(Requester.class,initComp);
3 }
```

Figure 9: Implementation of the method `startEnsemble()` in the class `TransferEnsemble`

```
1 protected void createComponents() {
2   Peer peer1 = new Peer(false, 0);
3   Peer peer2 = new Peer(true, 12345); ...
4   peer1.setNeighbor(peer2);
5   peer2.setNeighbor(peer3); ...
6   this.addComponent(peer1);
7   this.addComponent(peer2); ...
8 }
```

Figure 10: Instantiation of peers in the method `createComponents()` of the class `P2PSysManager`

```
1 public void startEnsembles() {
2   Ensemble ens1 = new TransferEnsemble("ens1");
3   this.addEnsemble(ens1);
4   ens1.startEnsemble(this.getComponent());
5
6   Ensemble ens2 = ...
7 }
```

Figure 11: Implementation of the `startEnsembles()` in the class `P2PSysManager`

4. 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 for writing ensemble specifications, 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 corresponds to the ensemble specification. Therefore, we propose the domain-specific language (DSL) HelenaText. The language provides a concrete syntax for ensemble specifications supporting roles and ensemble structures as first-class citizens. It is fully integrated into Eclipse with an editor including syntax highlighting, content assist, and validation. Moreover, we provide an automatic code generator which translates a HelenaText specification to jHelena code.

This section first gives an overview about the workflow for defining the HelenaText DSL and implementing the corresponding Eclipse plug-in (which provides the editor and the code generator). Afterwards, we describe how HelenaText realizes the Helena modeling approach and outline the generation rules of the code generator.

4.1. Workflow for the Implementation of the HelenaText Plug-In

For the development of the HelenaText DSL and the corresponding Eclipse plug-in, we rely on the Xtext workbench of Eclipse (`www.eclipse.org/Xtext/`). This workbench provides not only the possibility to define the DSL itself, but also supports the implementation of a customized editor, validator, and Java code generator in one workbench. Fig. 12 gives an overview of the steps that have to be taken for the implementation of the HelenaText plug-in with Xtext.

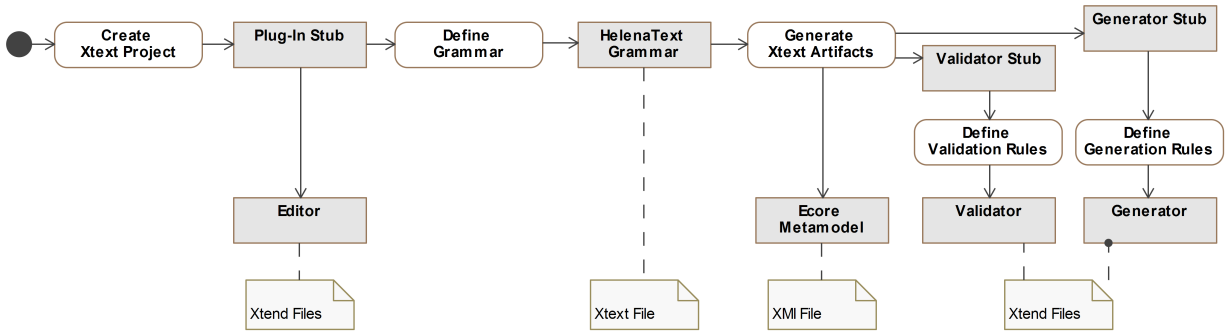


Figure 12: Workflow for the implementation of the HelenaText plug-in

In the first step, a new Xtext project is created. Amongst others, it has to be defined which file ending the desired DSL files should have. In our case, we chose the String `helena`. A simple Eclipse plug-in is created, that is already executable, but provides only a standard editor which accepts any text.

Together with the plug-in stub, an example grammar file is generated. In this Xtext file, we define the grammar that determines our new DSL HelenaText. The grammar rules in Xtext are described in detail in Sec. 4.2. Any ensemble specification defined by the user of the Eclipse plug-in later on has to conform to this grammar.

Afterwards, the command `Generate Xtext Artifacts` is triggered to validate the specified grammar for correctness. For the valid grammar, the metamodel (in form of an Ecore metamodel) as well as Java classes implementing the DSL are derived from the grammar. Moreover, XTEND classes which serve as stubs for the code generator and the validator are created.

The validator stub is then extended by validation rules for properties and context conditions of HELENA which could not be expressed in the grammar. The validation rules are written in XTEND and are described in detail in Sec. 4.2. When defining a concrete ensemble specification in HELENATEXT later on, the Eclipse plug-in will issue validation based on these rules to check that the specification satisfies the conditions required by HELENA.

The generator stub is extended to be able to transform concrete HELENATEXT ensemble specifications to executable jHELENA code. The generation rules are also written in XTEND and are described in detail in Sec. 4.3. The Eclipse plug-in will issue code generation based on these rules whenever a valid ensemble specification is created in the HELENATEXT plug-in later on.

The complete Eclipse plug-in for HELENATEXT consists of an HELENATEXT editor supporting the specified grammar and its validation rules as well as the code generator. To make it accessible to a user, the plug-in is exported using the Eclipse plug-in development assistant (e.g., as a jar file) and then imported into her Eclipse platform by the user.

4.2. The Domain-Specific Language HELENATEXT

For our domain-specific language HELENATEXT, we define the grammar in a BNF-like notation of XTEXT. The grammar follows the formal definitions of the HELENA modeling elements like component and role types, ensemble structures and role behaviors. Constraints which cannot be included into the DSL grammar are formulated as validation rules written in XTEND. The rules for all syntactic constructs of HELENATEXT can be found at [4]. Here, we only want to focus on two particular aspects.

Structural Aspects:. To exemplify the derivation of the grammar rules for types, let us revisit the definition of a role type from Sec. 2.1: a role type *rt* over a given set of component types CT is a tuple $rt = (rtnm, rtcomptypes, rtattrs, rtmsgs)$. Fig. 13 shows the corresponding grammar rule. A role type declaration in HELENATEXT must start with the keyword `roleType` followed by its `name` referring to *rtnm*. The set *rtcomptypes* of component types which can adopt the role are reflected by the list `compTypes` after the keyword `over`. It is a list of references to already defined component types which is expressed by the square brackets, the cross reference concept of XTEXT. In curly braces, the two sets `roleattrs` referring to *rtattrs* and `rolemsgs` referring to *rtmsgs* are defined in arbitrary order. Opposed to HELENA, we assume typed attributes and typed data parameters in HELENATEXT.

```
1  RoleType:
2  'roleType' name=ValidID 'over' compTypes+=[ComponentType](','compTypes+=[ComponentType])* '{'
3     (
4           roleattrs += ('roleattr' type=JvmTypeReference  name=ValidID ';')
5        |  rolemsgs  += ('rolemsg'  direction=MsgDirection name=ValidID
6                                    formalRoleParamsBlock=FormalRoleParamsBlock
7                                    formalDataParamsBlock=FormalDataParamsBlock ';')
8     )*
9  '}'
10 ;
```

Figure 13: XTEXT grammar rule for role types in HELENATEXT

However, the DSL grammar rule cannot express that the lists `compTypes`, `roleattrs`, and `rolemsgs` all have to be duplicate-free to represent the sets *rtcomptypes*, *rtattrs*, and *rtmsgs*. For that, a validation rule in XTEND is added (cf. Fig. 14). Each set of elements is handled separately, for messages we even split the set according to whether they are incoming or outgoing messages (cf. line 5-6). For each set, we call the method `findDuplicates` which reports an error in line 13 if an element with the same name exists in the investigated set.

The concrete syntax for the declaration of the role type `Router` of Fig. 1c is shown in Fig. 15.

```
1  @Check
2  def check_rt_hasDuplicates(RoleType rt) {
3    findDuplicates(rt.compTypes);
4    findDuplicates(rt.roleattrs);
5    findDuplicates(rt.rolemsgs.filter[direction == MsgDirection.OUT || direction == MsgDirection.INOUT]);
6    findDuplicates(rt.rolemsgs.filter[direction == MsgDirection.IN  || direction == MsgDirection.INOUT]);
7  }
8
9  private def void findDuplicates(Iterable<T extends AbstractDuplicateFreeObject> list) {
10   var Set<String> nameSet = new TreeSet();
11   for (AbstractDuplicateFreeObject elem : list.filterNull) {
12     if (!nameSet.add(elem.name)) {
13       error('Duplicate declaration of ' + elem.name, ...)
14     }
15   }
16 }
```

Figure 14: XTEND validation rule for role types in HELENATEXT

```
1 roleType Router over Peer {
2   rolemsg in/out reqAddr(Requester req)();
3   rolemsg out sndAddr(Provider prov)();
4 }
```

Figure 15: Router in the p2p example specified in HELENATEXT

Dynamic Behavior. The second step is to add dynamic behavior to complete the ensemble specification. For that purpose, we have to specify role behaviors which are given by process terms including actions for role instance creation and retrieval, sending and retrieving messages, operations calls, and setting values of attributes. The grammar rule for defining process terms (cf. Fig. 16) directly follows the inductive definition in Def. 2.

```
1  Process: 'process' name=ValidID '=' processTerm=ProcessTerm;
2
3  ProcessTerm:
4       {NilTerm} 'nil'
5    |  {ActionPrefix} (action=Action '.' processTerm=ProcessTerm)
6    |  {GuardedChoice}
7         ( 'if' '(' ifGuard   = Guard ')' 'then' '{' ifBranch   = ProcessTerm '}'
8           ('or' '(' orGuards += Guard ')' 'then' '{' orBranches+= ProcessTerm '}')* )
9    |  {ProcessInvocation} process=[Process]
10 ;
```

Figure 16: XTEXT grammar rule for process terms in HELENATEXT

In Sec. 2.2, we informally stated that role behaviors have to be well-formed, e.g., messages can only be sent by roles which declare them. Those well-formedness conditions cannot be expressed in the DSL grammar. Therefore, we add validation rules written in XTEND. Fig. 17 shows the validation of outgoing message calls. In line 3, we iterate over all declared messages of the corresponding role type of the role behavior where the outgoing message call was issued. If none of the messages has the same name as the called message (cf. line 4), we report an error in line 25. Otherwise, we check whether the message was declared outgoing in the role type in line 5, whether the actual parameters of the call match the formal parameters in line 9 and line 13, and whether the called message is allowed as incoming message at the receiving role in line 17.

In Fig. 18, we present the role behavior of a router specified with HELENATEXT.

4.3. Code Generator

To make HELENATEXT specifications executable, we provide an automatic code generator. The generator uses rules written in XTEND to translate HELENATEXT elements to jHELENA code. It takes a HELENATEXT file containing a particular ensemble specification as input and generates a package for the ensemble application which is split into two parts, the (sub)packages `src-gen` and `src-user`. The package `src-gen` is already complete and must not be touched anymore while the package `src-user` offers templates which must be implemented by the user.

```
@Check
def check_rb_messageCallFitsToRoleType(OutgoingMessageCall call) {
  for (roleMsg : call.parentRoleBehavior.roleTypeRef.rolemsgs) {
    if (call.msgName == roleMsg.name) {
      if (!call.directionMatches(roleMsg)) {
        error('The underlying role type has to allow sending/receiving the message.',...);
        return;
      }
      if (!call.roleParamsMatchInType(roleMsg)) {
        error('The role parameters do not fit to the ones of the message in the role type.',...);
        return;
      }
      if (!call.dataParamsMatchInType(roleMsg)) {
        error('The data parameters do not fit to the ones of the message in the role type.',...);
        return;
      }
      if (!call.communicationPartnerHasMatchingMsg()) {
        error('The message has to be allowed as incoming message at the receiver.',...);
        return;
      }
      // everything was ok
      return;
    }
  }
  error('The message has to be a message of this role type.',...);
}
```

Figure 17: XTEND validation rule for role types in HELENATEXT

```
roleBehavior Router = RouterProc {
  process RouterProc =
    ? reqAddr(Requester req)() .
    if ( owner.hasFile ) then { Provide }
    or (! owner.hasFile) then { Fwd }

  process Provide =
    prov <- create(Provider, owner) .
    req ! sndAddr(prov)() . nil

  process Fwd =
    if ( plays(Router, owner.neighbor) ) then { nil }
    or ( !plays(Router, owner.neighbor) ) then { Create }

  process Create =
    router <- create(Router, owner.neighbor) .
    router ! reqAddr(req)() .
    RouterProc
}
```

Figure 18: Router in the p2p example specified in HELENATEXT

For the p2p example, the generated package `p2p` is shown in Fig. 19. In comparison to Fig. 6 where we explained the implementation of the p2p example by hand, the package `p2p` is now split into two parts: the package `src-gen` contains only classes which could be completely generated from the HELENATEXT specification; the package `src-user` provides base classes where the user has to implement the parts which cannot be generated from the ensemble specification.

Package `src-gen`. Let us start with the classes of the package `src-gen`. It contains the generated subclasses for the abstract base classes of the `developer-interface`. These subclasses, like `Peer`, `Requester`, `Router`, and `Provider` correspond to the types of the given ensemble structure. They implement the structural composition of a `TransferEnsemble` as well as the dynamic behavior of all roles as explained in Sec. 3.3. The generated `P2PSysManager` implements the method `configureTypes()` as shown in Fig. 7. It takes care to create objects for the `metadata` classes which represent types and the ensemble structure in accordance with the p2p ensemble specification.

Package `src-user`. The package `src-user` provides generated implementation classes to realize the parts which were not specified in the HELENATEXT ensemble specification like the effect of component operations (i.e., the body of the

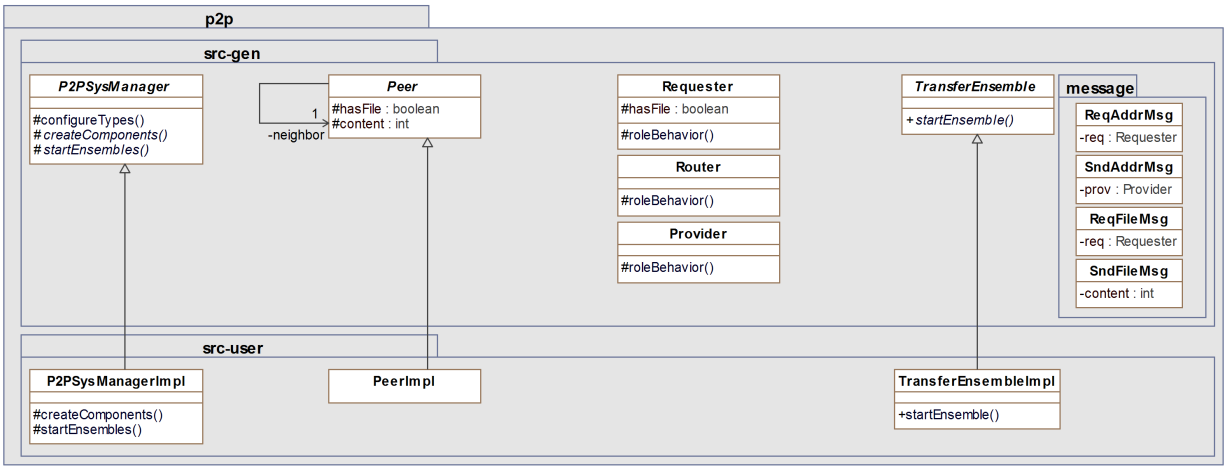


Figure 19: Generated jHELENA implementation for the p2p example

corresponding methods) and the initialization of ensembles. In the p2p example, components do not have any operations, thus we can omit their implementation which would be part of the class `PeerImpl`. However, to initialize the ensemble for file transfer the user needs to implement the two generated implementation classes `P2PSysManagerImpl` and `TransferEnsembleImpl`. They are generated with empty stubs for the methods `createComponents()`, `startEnsembles()`, and `startEnsemble()` resp. to be implemented by the user.

Generator Rules. The rules to generate the code for both packages are defined by template expressions in XTEND. Fig. 20 shows an excerpt of such an XTEND rule for the generation of the role classes. The operation `compile` is called for any role type given in a HELENATEXT specification and generates the corresponding class declaration in jHELENA. Basically anything in the operation `compile` is written to the generated class file except text enclosed in tag brackets «» which must be evaluated first. For example, in line 2 the class-header is built. The name of the class is dynamically evaluated from the expression «`it.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 Fig. 8 in Sec. 3.3). Afterwards, in line 4-6 of the XTEND rule, all attributes of the role type are generated as instance variables (which are none for the role type `Router`). Lines 7-9 declare additional instance variables for any parameters of incoming messages or created role instances 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`, we need instance variable to store the values of the two created role instances `prov` and `router` as well as of the parameter `req` of the incoming message `reqAddr` (cf. Fig. 8).

For the role behavior itself the method `roleBehavior` is generated from the process term representation in HELENATEXT (cf. line 15-18). Basically, the process term is generated by calling the function `compileProcessTerm`. Similarly to the formal definition of role behaviors in Def. 2, this function is inductively defined. Here, we only show the translation of action prefix in line 27-30. The generated method `roleBehavior` for the router is shown in Fig. 8.

Lastly, we also generate getters and setters for all attributes of the considered role type and its owning component types (cf. line 20-23). These methods are not shown in Fig. 8 since they are standard implementations.

Similarly, XTEND rules for all concepts of HELENA are defined such that executable jHELENA code can be generated from a HELENATEXT specification. The full set of generator rules can be found at [4].

```
1  def compile(RoleType it){
2  '''public class «it.classname» extends Role {
3
4     «FOR field : it.roleattrs»
5       protected «field.type» «field.name»;
6     «ENDFOR»
7     «FOR inst : rb.abstractInstances»
8       protected «inst.type» «inst.name»;
9     «ENDFOR»
10
11    public «it.classname»(Ensemble ens, Integer capacity)  throws ... {
12      super(ens, capacity);
13    }
14
15    @Override
16    protected void roleBehavior() throws ... {
17      «compileProcessTerm(rb.compileRoleBehavior)»
18    }
19    ...
20    «FOR field: allFields»
21      private «field.type» «field.getterName»() {...}
22      private void «field.setterName»(«field.type» «field.name») {...}
23    «ENDFOR»
24    }'''
25 }
26
27 def compileProcTerm(ActionPrefix actionPrefix){
28 '''«compileAction(actionPrefix.action)»
29    «compileProcTerm(actionPrefix.processTerm)»'''
30 }
31 ...
```

Figure 20: XTEND generation rule for role types in HELENATEXT

5. Conclusion

In this paper, we presented HELENATEXT, a domain-specific language to specify ensembles, and jHELENA, a Java framework to realize and execute ensemble specifications. HELENATEXT relies on the XTEXT workbench of Eclipse and therefore provides an Eclipse plug-in with an editor which gives the user full content assist for writing ensemble specifications and checks the user-defined models for validity according to the formal HELENA definitions. Additionally, a code generator was implemented which translates ensemble specifications in HELENATEXT to executable jHELENA code. jHELENA transfers the concepts of roles and ensembles to an object-orientated platform and directly implements the formal foundations and the execution model of the HELENA approach. The use of HELENATEXT and its code generator (and thus jHELENA) was demonstrated at a p2p example throughout the paper.

5.1. Related Work

Ensemble-Based Systems. The EU project ASCENS [5, 6] 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 [7, 8] 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. In contrast, HELENA relies on message exchange between participants of ensembles and introduces a second role layer on top of a component-based platform to allow a more flexible mechanism for dynamic ensemble composition. DEECo [9] 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. However, HELENA is more concrete since we include explicit notions of interaction and collaboration. Related approaches have been developed in the context of multi-agent systems and multi-party session types, for instance in the Scribble framework [10]. However, none of these methods formalizes concurrent execution of ensembles which is built-in in our ensemble automata[5].

[5]For a more detailed comparison of the HELENA ideas with the literature see [1].

Roles. With HELENA, we offer a rigorous modeling method for describing task-oriented groups. Modeling evolving objects with roles as perspectives on objects has been proposed by various authors [11, 12, 13, 14], but they do not see them as autonomic entities with behavior as we do in HELENA. Steegmans et al. [15] propose a role model where agents commit themselves to roles and therefore execute the associated behavior given by action diagrams. However, they do not transfer the idea of roles to the implementation level as we do it with jHELENA, but rather rely on free-flow architectures for realization. For describing dynamic behaviors, we share ideas with different process calculi [7, 16], but we introduce dynamic instance creation for roles on selected components[5].

Implementation Frameworks. The idea to describe structures of interacting objects without having to take the entire system into consideration was already introduced by several authors [17, 18, 19, 20], but they do not consider roles as autonomic entities and do not tackle concurrently running ensembles as we do in HELENA. The modeling approach Macodo [21] introduces a set of role-based abstractions to define collaborations. It is supported by a proof-of-concept middleware which provides appropriate programming concepts to map the role-based abstractions to Web service technologies. However, their focus is only on the collaboration-level and does not include the concrete realization of individual role behaviors.

5.2. Future Work

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 are currently investigating how to check ensemble specifications for goal satisfaction [22]. Thereby, we rely on LTL formulae to express goals like proposed in [23, 24]. We generate PROMELA code from ensemble specifications which can be validated for the satisfaction of goals with the model-checker Spin [25]. Lastly, we plan to examine requirements for collaboration correctness and to integrate tools for the analysis of ensemble specifications to check the absence of collaboration errors.

References

[1] R. Hennicker, A. Klarl, Foundations for Ensemble Modeling - The Helena Approach, in: Specification, Algebra, and Software, Vol. 8373 of LNCS, Springer, 2014, pp. 359–381.

[2] A. Klarl, L. Cichella, R. Hennicker, From Helena Ensemble Specifications to Executable Code, in: International Symposium on Formal Aspects of Component Software, Vol. 8997 of LNCS, Springer, 2015, pp. 183–190.

[3] A. Klarl, R. Hennicker, Design and Implementation of Dynamically Evolving Ensembles with the Helena Framework, in: Australasian Software Engineering Conference, IEEE, 2014, pp. 15–24.

[4] A. Klarl, R. Hennicker, The Helena Framework (2015).
URL `http://www.pst.ifi.lmu.de/Personen/team/klarl/helena`

[5] The ASCENS Project (2015).
URL `http://www.ascens-ist.eu`

[6] M. Wirsing, M. Hölzl, M. Tribastone, F. Zambonelli, ASCENS: Engineering Autonomic Service-Component Ensembles, in: B. Beckert, F. Damiani, M. Bonsangue, F. de Boer (Eds.), 10th International Symposium on Formal Methods for Components and Objects, Vol. 7542 of Lecture Notes in Computer Science, Springer, 2012.

[7] R. De Nicola, G. L. Ferrari, M. Loreti, R. Pugliese, A Language-Based Approach to Autonomic Computing, in: B. Beckert, F. Damiani, F. S. de Boer, M. M. Bonsangue (Eds.), 10th International Symposium on Formal Methods for Components and Objects, Vol. 7542 of Lecture Notes in Computer Science, Springer, 2011, pp. 25–48.

[8] R. De Nicola, M. Loreti, R. Pugliese, F. Tiezzi, SCEL: a Language for Autonomic Computing, Tech. rep., IMT, Institute for Advanced Studies Lucca, Italy (2013).

[9] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, F. Plasil, DEECo: an Ensemble-based Component System, in: Proceedings of 16th International Symposium on Component-Based Software Engineering, ACM, 2013, pp. 81–90.

[10] N. Yoshida, R. Hu, R. Neykova, N. Ng, The Scribble Protocol Language, in: M. Abadi, A. L. Lafuente (Eds.), 8th International Symposium Trustworthy Global Computing, Vol. 8358 of Lecture Notes in Computer Science, Springer, 2013, pp. 22–41.

[11] G. Gottlob, M. Schrefl, B. Röck, Extending Object-Oriented Systems with Roles, ACM Trans. Inf. Syst. 14 (3) (1996) 268–296.

[12] B. B. Kristensen, K. Østerbye, Roles: Conceptual Abstraction Theory and Practical Language Issues, TAPOS 2 (3) (1996) 143–160.

[13] F. Steimann, On the representation of roles in object-oriented and conceptual modelling, Data Knowl. Eng. 35 (1) (2000) 83–106.

[14] F. Steimann, Formale Modellierung mit Rollen, Habilitation Thesis, Universität Hannover (2000).

[15] E. Steegmans, D. Weyns, T. Holvoet, Y. Berbers, A Design Process for Adaptive Behavior of Situated Agents, in: International Conference on Agent-Oriented Software Engineering, Springer, 2005, pp. 109–125.

[16] P.-M. Deniélou, N. Yoshida, Dynamic Multirole Session Types, in: Symposium on Principles of Programming Languages, ACM, 2011, pp. 435–446.

[17] 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, NODe '02, Springer-Verlag, London, UK, UK, 2003, pp. 248–264.

[18] M. Baldoni, U. Studi, T. Italy, Interaction between Objects in powerJava, Journal of Object Technology 6 (2007) 7–12.
[19] T. Reenskaug, Working with objects: the OOram Framework Design Principles, Manning Publications, Greenwich, CT, 1996.
[20] T. Tamai, N. Ubayashi, R. Ichiyama, Objects as Actors Assuming Roles in the Environment, in: Software Engineering for Multi-Agent Systems V, Vol. 4408 of LNCS, Springer, 2007, pp. 185–203.
[21] R. Haesevoets, D. Weyns, T. Holvoet, Architecture-Centric Support for Adaptive Service Collaborations, Transaction on Software Engineering Methodology 23 (2014) 2:1–2:40.
[22] R. Hennicker, A. Klarl, M. Wirsing, Model-Checking Helena Ensemble Specifcations with Spin, in: Logic, Rewriting, and Concurrency - Festschrift Symposium in Honor of José Meseguer, LNCS, Springer, submitted.
[23] A. van Lamsweerde, Requirements Engineering: From System Goals to UML Models to Software Specifications, Wiley, 2009.
[24] M. B. Dwyer, G. S. Avrunin, J. C. Corbett, Patterns in Property Specifications for Finite-state Verification, in: International Conference on Software Engineering, ACM, 1999, pp. 411–420.
[25] G. Holzmann, The Spin Model Checker: Primer and Reference Manual, Addison-Wesley, 2003.