HAL open science

Role-Based Development of Dynamically Evolving Esembles

Rolf Hennicker

▶ To cite this version:

Rolf Hennicker. Role-Based Development of Dynamically Evolving Esembles. 24th International Workshop on Algebraic Development Techniques (WADT), Jul 2018, Egham, United Kingdom. pp.3-24, 10.1007/978-3-030-23220-7_1 . hal-02364578

HAL Id: hal-02364578

https://inria.hal.science/hal-02364578v1

Submitted on 15 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Role-based Development of Dynamically Evolving Esembles

Rolf Hennicker[1]

Ludwig-Maximilians-Universität München, Germany
hennicker@ifi.lmu.de

Abstract. An ensemble is a set of computing entities that collaborate to perform a certain task. Typically an ensemble changes dynamically its constitution such that new members can join and other members can leave an ensemble during its execution. The members of an ensemble interact through message exchange. They are modelled as instances of certain role types which can be adopted by components of an underlying component system. We propose a dynamic logic to describe the evolution of ensembles from a global perspective. Using the power of dynamic logic with diamond and box modalities over regular expressions of actions (involving role instance creation, message exchange and component access) we can specify safety and liveness properties as well as desired and forbidden interaction scenarios. Thus our approach is suitable to write formal requirements specifications for ensemble behaviours. For ensemble design and implementation we propose ensemble realisations. An ensemble realisation takes a local view by giving a constructive specification for each single role type in terms of a process algebraic expression. Correctness of an ensemble realisation is defined semantically: its generated ensemble transition system must be a model of the requirements specification. We consider bisimulation of ensemble transition systems and show that our approach enjoys the Hennessy-Milner property.

Keywords: ensemble, distributed system, component, role, dynamic logic, interaction scenario, bisimulation equivalence

1 Introduction

Autonomic computing and global interconnectedness of nodes allow the dynamic formation of collective systems which pose new challenges to software engineers. Typically autonomic nodes have the ability to perceive their environment and adapt their behaviour accordingly. They interact with other nodes in the system to collaborate in teams for some global goal. Such teams are called ensembles in the EU project ASCENS [15,16]. We claim that well-known techniques, like component-based software engineering, are not sufficient for modelling ensembles but must be augmented with other features to deal with the particular characteristics of ensembles. As a framework for rigorous ensemble development we have proposed the two-layer approach Helena [7,12] whose design is motivated by the

following considerations. While a component model describes the architecture of a target system, ensembles are dynamically formed on demand as specific, goal-oriented communication groups running on top of a target system. In particular, different ensembles may run concurrently on the same target system (dealing with different tasks). In Helena the target platform is component-based, but it is crucial to recognise that the same component instance may take part in different ensembles under particular, ensemble-specific *roles*. Thus a role is an abstraction of the part that an individual component plays in a specific collaboration. Focusing on roles allows us to put different views on a component and to concentrate on those capabilities needed for the execution of a particular ensemble. Ensemble modelling it is then the task to identify which roles are needed in an ensemble, which components could play the roles, and how role instances communicate when pursuing a common goal.

Helena introduces roles as first-class artefacts in ensemble models and implementations. Components form the basic layer. Component instances are passive objects (of long term nature) storing data and providing operations which implement certain functionalities, mostly related to the components' data. Components have a simple life-cycle; their operations can be called at any time. Ensembles, however, evolve dynamically which may involve ensemble extensions when components join an ensemble under some role. Role instances are active entities (i.e. processes), often of short term nature, having role-specific communication abilities (in the form of input and output messages) and running concurrently on top of their owning components. Role instances can access the operations of their owning component by operation calls. Components do not talk to each other; any kind of collaboration is performed by message exchange between the active role instances played by components. Thus, by separating roles and components, we get a two-layered approach which decouples communication (performed between roles) and computation (performed by components). Moreover, designing a behaviour for each role of a component (which is implemented by a single thread per role) is much simpler then designing a complex life-cycle for a component which then must integrate all the different tasks a component might be involved. A prototypical, two-layered implementation framework for Helena models is described in [14,13]. The two-layered approach supports also adaptation of components by switching between roles as discussed in [11].

Constructive descriptions of ensemble-based systems are supported by the high-level programming language SCEL [3] for autonomic systems, by the component-based DEECo framework [1] and by Helena where process-algebraic expressions are provided for describing role behaviours. A more abstract level is considered in [9] where, similarly to "classical" top down methodologies, the development of an ensemble-based system starts with a property-oriented specification and only later a concrete realisation is constructed, which can then be checked for correctness. The approach in [9] deals, however, only with ensembles of processes and does not take into account components and the roles they play. The goal of the current paper is to provide an appropriate extension which can be used for a top down development of ensembles in accordance with the

component and role layers of HELENA. Therefore, in contrast to [9], this paper incorporates the two-layer approach which requires significant syntactic and semantic extensions of [9]. On the syntactic side we integrate component system signatures, their connection to ensemble signatures (in terms of the "canPlay" relations) and we have new actions in the logic for component access as well as actions with "wild cards". The latter allow us, in particular, to express safety and liveness properties. The semantic notions and results of [9] are extended accordingly. In particular, we define semantic component models on top of which semantic structures for ensembles are constructed.

In our approach specifications describe *global* properties of collaborations performed by an ensemble. They are written in a dynamic logic style [4]. This allows us to focus, in contrast to temporal logics, on explicit interactions and complex interaction scenarios which are typical for a certain ensemble. The logic uses diamond and box modalities equipped with regular expressions of actions, like sequential composition and iteration. Atomic actions are either interactions - in the form of message exchange between two role instances - or the creation of a new role instance on top of a component (i.e. the specified component instance joins an ensemble under a particular role), or component access performed by a role instance. Additionally we allow quantification over role instances. Using the power of dynamic logic we can thus specify desired and forbidden interaction scenarios. Hence, our approach is suitable to write formal requirements specifications for global, complex interaction behaviours.

Semantic structures of our logic are ensemble transition systems. They have two layers: a transition system for the underlying component system and a transition system describing the execution of an ensemble on top of it, in particular involving all the role instances played by component instances. The semantics of an ensemble specification is given by the class of its models, i.e. by all ensemble transition systems which satisfy the axioms of the specification. Thus we support loose specification and underspecification. A refinement relation between ensemble specifications is defined by model class inclusion. We define a bisimulation relation between ensemble transition systems and show that the validity of ensemble sentences is preserved by ensemble bisimulation. Hence, the semantics of an ensemble specification is closed under bisimulation equivalence. Moreover, for image-finite ensemble transition systems the validity of the same sentences implies bisimulation; thus the Hennessy-Milner property holds.

In the last part of this work, we consider ensemble realisations and a formal correctness notion. An ensemble realisation takes a *local* view and specifies, as in a HELENA design model, a behaviour for each single role type in terms of a process algebraic expression. All instances of the type must respect the prescribed behaviour. We show how a (global) ensemble transition system can be generated from the local behaviours of role instances. An ensemble realisation is correct, if its generated ensemble transition system satisfies the (logical) sentences of the specification. In particular, two (bisimulation) equivalent ensemble realisations implement the same specifications.

The paper is organised as follows: Section 2 defines the syntactic notions of ensemble specifications based on ensemble signatures and sentences. In Section 3 we consider ensemble transition systems used for the semantics of ensemble specifications and we show the invariance of sentences under ensemble bisimulation and the Hennessy-Milner property. Then, in Section 4, we study correct ensemble realisations. Concluding remarks are given in Section 5.

2 Ensemble Specifications

Our approach relies on a strict separation of syntax and semantics, in particular between types and their instances. In this section we consider component types, role types, ensembles signatures, ensemble formulas and specifications. To get an intuition we will sometimes also refer to component and role instances to be introduced later, in Section 3. In the following we assume given a not further specified set of data types.

Component types. Components form the basic layer of our approach. To classify components we use component types. A *component type* $ct = (ctnm, attrs, opns)$ has a name $ctnm$ and declares a set of attributes $attrs$ to store information and a set of operations $opns$ which can be exploited by the roles of components. We write $\mathtt{opns}[ct]$ for the operations of ct. An *attribute* $t\ a$ has a name a and a type t. An attribute type is either a data type or a component type such that an attribute value can point to a component instance. In this case the attribute is called *reference attribute*. An *operation* is of the form $opnm(fparams)$ or $t\ opnm(fparams)$, the former being a *pure operation* and the latter an *operation with result*. $opnm$ is the name of the operation, $fparams = t_1\ p_1, \ldots, t_n\ p_n$ is a list of typed formal parameters and t a result type. The types $t, t_1, \ldots, t_n$ of an operation are data types not further specified here.

Component system signature. A *component system signature* $C\Sigma$ is a set of component types whose reference attributes use only component types in $C\Sigma$. We write $\mathtt{opns}[C\Sigma]$ for the set of all operations used in the component types of $C\Sigma$. A component system signature $C\Sigma$ is *finite* if it has finitely many component types each with finitely many attributes and operations.

Role types. For performing certain tasks, components team up in ensembles. Each participant in the ensemble contributes specific functionalities to the collaboration; we say, the participant plays a certain *role* in the ensemble. To classify roles we use role types. A *role type* $rt = (rtnm, mts_{in}, mts_{out})$ has a name $rtnm$ and sets mts_{in} and mts_{out} of input and output message types respectively, which model the interaction capabilities provided by each instance of a role type. We write $\mathtt{mts}_{in}[rt]$ for mts_{in}, $\mathtt{mts}_{out}[rt]$ for mts_{out}, and $\mathtt{mts}[rt]$ for $mts_{in} \cup mts_{out}$. A *message type* is of the form $mtnm(fparams)$ where $mtnm$ is a message type name and $fparams = t_1\ p_1, \ldots, t_n\ p_n$ is a list of typed formal parameters. Here the parameter types $t_1, \ldots, t_n$ are either role types or data types.

Ensemble Signature. Let $C\Sigma$ be a component system signature. An *ensemble signature* $E\Sigma = (rtypes, canPlay)$ *over* $C\Sigma$ consists of a set *rtypes* of role types and a surjective relation $canPlay \subseteq C\Sigma \times rtypes$. This relation indicates that, whenever $(ct, rt) \in canPlay$, then any component instance of type ct can potentially play the role rt (being represented later on by a role instance of type rt). We write $\texttt{rtypes}[E\Sigma]$ for *rtypes*, $\texttt{canPlay}[E\Sigma]$ for *canPlay* and $\texttt{mts}[E\Sigma]$ for the set $\bigcup_{rt \in \texttt{rtypes}[E\Sigma]} \texttt{mts}[rt]$ of all message types used in the role types of $E\Sigma$. We assume that these message types use in their parameter lists, besides data types, only role types in *rtypes*. In this work we consider only closed systems where $\bigcup_{rt \in \texttt{rtypes}[E\Sigma]} \texttt{mts}_{in}[rt] = \bigcup_{rt \in \texttt{rtypes}[E\Sigma]} \texttt{mts}_{out}[rt]$. An ensemble signature $E\Sigma$ is *finite* if it has finitely many role types each with finitely many message types.

Example 1. Throughout this paper we consider a (simplified version of a) file transfer ensemble which runs on a peer-2-peer network supporting the distributed storage of files that can be retrieved upon request. Several peer components work together to request and transfer a file: One peer plays the role of a `Requester` of the file, other peers act as `Router`s and the peer storing the requested file adopts the role of a `Provider`. The component type `Peer` models peers. Each peer stores a file in the corresponding attribute and it has a neighbour whose identity is stored in the corresponding reference attribute. Peers provide operations to get files and to store files. Each kind of role is modelled by a role type whose instances can be created and run on the peer components. The idea of the collaboration is that a requester issues a request for the address of a provider of a certain file identified by a string (message type `reqAddr(Requester r, String s)`). This address request is forwarded by routers through the network until a provider is found. Then the provider address is sent from the last active router to the requester (`sndAddr(Provider p)`). Finally, the requester asks the provider for the file (`reqFile(Requester r, String s)`) which is then sent to the requester (`sndFile(File f)`). It may also happen that no appropriate provider is found. In this case a router sends a notification to the requester (`notFound()`). The ensemble signature of the file transfer ensemble is graphically presented in Fig. 1. The directions of the message type arrows indicate for which role types a message type is input or output or both. Note that for `Router` the `reqAddr` message type is input *and* output since routers may forward address requests to other routers. □

An ensemble specification describes static and dynamic properties of a system of collaborating entities. The static aspects are represented by an ensemble signature. For the dynamic properties a specification takes a global view of an ensemble focusing on the desired (and not desired) interactions between the participants of an ensemble, on the creation of new ensemble members and on component access. To specify collaborations we use atomic actions and composed (structured) actions formed by sequential composition (;), union (+) and iteration (*) borrowed from dynamic logic [4].

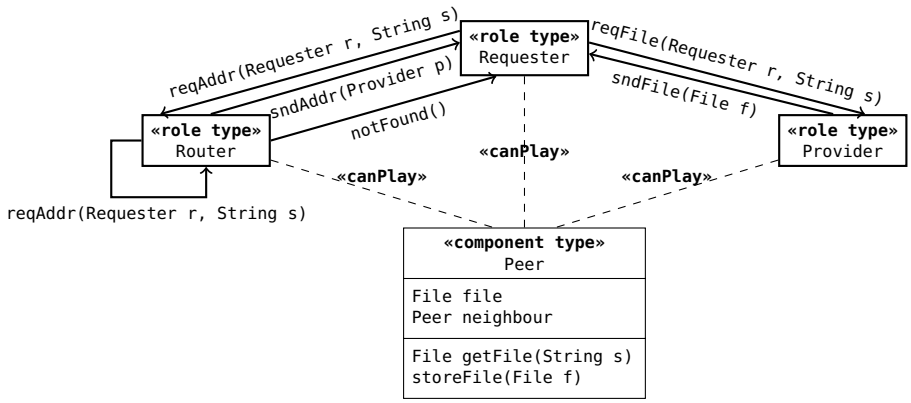

Fig. 1: Ensemble signature for the file transfer ensemble

In the following let $E\Sigma$ be a finite ensemble signature over a finite component system signature $C\Sigma$. We assume given countably infinite sets $RVar$ of role instance variables and $DVar$ of data variables.

Actions. Four kinds of *atomic actions* are distinguished where p, r are variables in $RVar$ and $d \in DVar$ is a data variable.

(a) A create action $p := r.\mathbf{create}(rt, ce)$ describes when a role instance denoted by r creates a new role instance of role type rt played by a component instance denoted by the component expression ce. The new role instance is assigned to variable p. The component expression can either be $r.\mathbf{playedBy}$ or $r.\mathbf{playedBy}.a$ where a is a reference attribute. Let us remark that we do not introduce an explicit action for deleting a role instance. A component simply gives up a role if its role instance becomes inactive.

(b) A communication action $(r \to p).mtnm(e_1, \ldots, e_n)$ describes when a role instance denoted by r sends a message with name $mtnm$ to a role instance denoted by p transmitting the values of the actual parameter expressions $e_1, \ldots, e_n$. These expressions are either role instance variables or data type expressions not further detailed here.

(c) A component access action $r.\mathbf{playedBy}.opnm(e_1, \ldots, e_n)$ describes when a role instance denoted by r accesses its owning component instance calling an operation with name $opnm$ and handing over the values of the actual parameter expressions $e_1, \ldots, e_n$. These expressions are data type expressions. Component access actions can also have the form $t\ d := r.\mathbf{playedBy}.opnm(e_1, \ldots, e_n)$ if the operation delivers a result which is then bound to variable d of data type t.

(d) A variable assignment $r := p$ assigns the role instance denoted by p to variable r.

We also allow generalisations of the atomic actions of type (a) - (c) where the variables p, r, d and expressions $ce, e_1, \ldots, e_n$ are partly or fully replaced by

a "wild card" represented by the special name **any**. Generalised create actions are, for instance, $\mathbf{any} := r.\mathbf{create}(rt, ce)$, $p := \mathbf{any}.\mathbf{create}(rt, \mathbf{any})$, or $\mathbf{any} := \mathbf{any}.\mathbf{create}(rt, \mathbf{any})$. The idea is that when **any** is used the particular instance involved in the action at the position of **any** is not relevant. An atomic action is called *fully generalised* if it has **any** at all possible positions.

The set $Act(E\Sigma, C\Sigma)$ of (structured) *actions over $E\Sigma$ and $C\Sigma$* is defined by the grammar

$$\alpha ::= a \mid \alpha;\alpha \mid \alpha + \alpha \mid \alpha^*$$

where a is a, possibly generalised, atomic action[1]. The sets $\mathrm{FV}(\alpha)$ of free variables and $\mathrm{BV}(\alpha)$ of bound variables of an action α are defined as expected where binding of a variable p to a role type rt can only happen via a create action $p := r.\mathbf{create}(rt, ce)$ and binding of a data variable d to a data type t can only happen via a component access action $t\,d := ce.opnm(...)$.

Shorthand notations. For a set $a_1, \ldots, a_n$ of fully generalised atomic actions we write $\{a_1, \ldots, a_n\}$ to denote the composed action $a_1 + \ldots + a_n$. We write $-\{a_1, \ldots, a_n\}$ to denote the composed action obtained from the union of all generalised atomic actions apart from those in $\{a_1, \ldots, a_n\}$. Note that this union is finite since there are only finitely many generalised atomic actions. This is due to the finiteness assumption for the underlying ensemble and component signatures. We write **allAct** for the composed action obtained by the union of all generalised atomic actions. It captures all actions that are semantically possible in an ensemble transition system; see below.

Ensemble formulas and sentences. Besides the usual propositional logic constructs ensemble formulas can compare the identity of role instances, they can be a modal formula with (composed) action α or they can be existentially quantified. The set $\mathrm{Fm}(E\Sigma, C\Sigma)$ of *formulas over $E\Sigma$ and $C\Sigma$* is defined by the following grammar

$$\varphi ::= \mathbf{tt} \mid r = p \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle\alpha\rangle\varphi \mid \exists r{:}rt.\varphi$$

where $\alpha \in Act(E\Sigma, C\Sigma)$ and $r, p \in RVar$. The set $\mathrm{FV}(\varphi)$ of free variables of a formula φ is defined as expected where binding of variables can happen by $\mathrm{FV}(\langle\alpha\rangle\varphi) = \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\varphi) \setminus \mathrm{BV}(\alpha))$ and $\mathrm{FV}(\exists r{:}rt.\varphi) = \mathrm{FV}(\varphi) \setminus \{r\}$.

A *sentence over $E\Sigma$ and $C\Sigma$* is a formula φ without free variables, i.e. $\mathrm{FV}(\varphi) = \emptyset$. The set of sentences over $E\Sigma$ and $C\Sigma$ is denoted by $\mathrm{Sen}(E\Sigma, C\Sigma)$. An *initialisation sentence* is a sentence φ, that does not contain any modality $\langle\alpha\rangle$. The set of initialisation sentences is denoted by $\mathrm{ISen}(E\Sigma, C\Sigma)$. We use the usual abbreviations: $\mathbf{ff} = \neg\mathbf{tt}, r \neq p = \neg(r = p), \varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi), [\alpha]\varphi = \neg\langle\alpha\rangle\neg\varphi, \forall r{:}rt.\varphi = \neg\exists r{:}rt.\neg\varphi$.

Let us note that by using generalised atomic actions and, in particular, the **allAct** notation, we can specify safety and liveness properties. Safety properties are expressed by sentences of the form $[\mathbf{allAct}^*]\varphi$. In particular, deadlock freeness is expressed by $[\mathbf{allAct}^*]\langle\mathbf{allAct}\rangle\mathbf{tt}$. Liveness properties, like "whenever an

[1] We could also add tests, as in dynamic logic, but we omit them for simplification.

action a has happened, an action b can eventually occur", can be expressed by $[\mathbf{allAct}^*; a]\langle\mathbf{allAct}^*; b\rangle\mathbf{tt}$.

Definition 1 (Ensemble specification). *Let $C\Sigma$ be a component system signature. An* ensemble specification *over $C\Sigma$ is a triple $EnsSpec = (E\Sigma, \Phi, \phi_0)$ where $E\Sigma$ is an ensemble signature over $C\Sigma$, $\Phi \subseteq \mathrm{Sen}(E\Sigma, C\Sigma)$ is a set of sentences, called* axioms *of $EnsSpec$, and $\phi_0 \in \mathrm{ISen}(E\Sigma, C\Sigma)$ is an* initialisation axiom.

Example 2. We formulate an ensemble specification for the file transfer ensemble introduced in Example 1. Concerning the starting condition of an ensemble the specification requires that in the initial ensemble state there is one role instance of type `Requester` and no role instances for any other type. The initialisation axiom ϕ_0 is

$$(\exists \mathrm{req{:}Requester}.\forall \mathrm{req'{:}Requester}.\mathrm{req'} = \mathrm{req}) \wedge$$
$$\neg\exists \mathrm{rout{:}Router}.\mathbf{tt} \wedge \neg\exists \mathrm{prov{:}Provider}.\mathbf{tt}$$

For ensemble execution we first require two abstract properties: (a) "whenever a file has been requested, it can eventually be sent" and (b) "a file cannot be sent if it hasn't been requested before". Using generalised atomic actions properties (a) and (b) can be expressed by the following two sentences φ_1 and φ_2 respectively:

$$\varphi_1 = [\mathbf{allAct}^*; \mathrm{reqFile}]\langle\mathbf{allAct}^*; \mathrm{sndFile}\rangle\mathbf{tt}$$
$$\varphi_2 = [(-\mathrm{reqFile})^*; \mathrm{sndFile}]\mathbf{ff}$$

where reqFile stands for $(\mathbf{any} \to \mathbf{any}).\mathrm{reqFile}(\mathbf{any}, \mathbf{any})$ and sndFile stands for $(\mathbf{any} \to \mathbf{any}).\mathrm{sndFile}(\mathbf{any})$.

At next, we are interested in the feasibility of a primary scenario that describes a successful execution of a file request. A requester `req` starts the collaboration by creating a router role (on its connected peer) and asks the router for the address of a provider (see α below). Arbitrarily, but finitely many routers forward the request by creating a next router on their neighbour component (see β^* below). The last router (whose owning component has the file) creates a provider role on its component and sends the address of the provider to the requester (see γ below). Then the requester asks the provider for the file, the provider gets the file by accessing its peer component, sends the file to the requester and, finally, the requester stores the file on its peer component (see δ below). The primary scenario is specified by the sentence

$$\varphi_3 = \forall \mathrm{req{:}Requester}.\langle\alpha; \beta^*; \gamma; \delta; \rangle\mathbf{tt}$$

where

$\alpha =$
```
rout:=req.create(Router,req.playedBy.neighbour);
(req→rout).reqAddr(req,"song")
```
$\beta =$
```
rout':=rout.create(Router,rout.playedBy.neighbour);
```

```
(rout→rout').reqAddr(req,"song");
rout:=rout'
```

$\gamma =$

```
prov:=rout.create(Provider,rout.playedBy);
(rout→req).sndAddr(prov)
```

$\delta =$

```
(req→prov).reqFile(req,"song");
song:=prov.playedBy.getFile("song");
(prov→req).sndFile(song);
req.playedBy.storeFile(song)
```

φ_3 allows iterations of arbitrary (but finite) length for forwarding the request to newly created router roles until a provider is found. Actually, the formula is still of abstract nature and does not really determine when a router component stores the requested file and therefore will adapt the role of a provider. This could, however, be easily done if we would extend our logic by tests. Moreover, let us remark that we have assumed a fixed name of the requested file. This could be avoided if we would go to open systems where the requester first would expect an input of an arbitrary file name from the environment captured by a formal parameter of the input message.

An alternative (secondary) scenario showing the possibility of a non successful delivery is specified by the sentence

$$\varphi_4 = \forall \text{req:Requester}.[\alpha;\beta^*](\langle\gamma;\delta\rangle\mathbf{tt} \wedge \langle\mu\rangle\mathbf{tt})$$

where $\mu =$ `(rout→req).notFound()`

The sentence φ_4 requires that during a routing phase a successful delivery of the file is possible and it is also possible that a requester is informed by the current router that no provider is found. □

3 Semantics of Ensemble Specifications and Bisimulation

For the semantic interpretation of ensemble specifications we use ensemble transition systems. Analogously to the two syntactic levels of components and ensembles, also the semantics is based on two levels.

We start by considering semantic models for a component system signature $C\Sigma$. A *component system state* over $C\Sigma$ is a pair $c\sigma = (cinsts, cdata)$ where $cinsts = \bigcup_{ct \in C\Sigma} cinsts_{ct}$ is the disjoint union of finite sets $cinsts_{ct}$ of (identifiers for) currently existing component instances of component type ct - similarly to a heap in object-oriented systems - and $cdata$ is a function assigning to each component instance $ci \in cinsts_{ct}$ a valuation of the attributes declared in ct. We do not further detail attribute valuations here, but we assume that whenever $a : ct'$ is a reference attribute of ct then $cdata(ci)(a) \in cinsts_{ct'}$ for each $ci \in cinsts_{ct}$. If $c\sigma = (cinsts, cdata)$ we write $\texttt{cinsts}[c\sigma]$ for $cinsts$. The set of component system states over $C\Sigma$ is denoted by $States(C\Sigma)$.

Component system models over $C\Sigma$, called $C\Sigma$-models, are labelled transition systems whose transitions express state changes caused by the execution of component operations. Thus the labels are *component access labels* of the form $ci.opnm(v_1, \ldots, v_n)$ expressing that the operation with name *opnm* is called on the component instance *ci* with data values $v_1, \ldots, v_n$. If the operation delivers a result value v, this will also be recorded on the transition by the notation $v = ci.opnm(v_1, \ldots, v_n)$. The set of labels over $C\Sigma$ is denoted by $Lab(C\Sigma)$.

$C\Sigma$-models constrain the use of labels on transitions by appropriate pre- and postconditions like "a component access on a component instance *ci* can only be executed if *ci* exists in the source state of the transition".

Definition 2 ($C\Sigma$-models). *Let $C\Sigma$ be a component system signature. A $C\Sigma$-*model *is a labelled transition system $M = (States(C\Sigma), c\sigma_0, Lab(C\Sigma), \rightarrow_M)$ such that*

- *$c\sigma_0 \in States(C\Sigma)$ is the initial component system state,*
- *$\rightarrow_M \subseteq States(C\Sigma) \times Lab(C\Sigma) \times States(C\Sigma)$ is a deterministic*[2] *transition relation such that for all $c\sigma \xrightarrow{ci.opnm(v_1,\ldots,v_n)}_M c\sigma'$ ($c\sigma \xrightarrow{v=ci.opnm(v_1,\ldots,v_n)}_M c\sigma'$ resp.) the following well-formedness conditions are satisfied:*
  (pre) there exists $ct \in C\Sigma$ such that $opnm(t_1\, p_1, \ldots t_n\, p_n) \in \mathtt{opns}[ct]$ ($t\, opnm(t_1\, p_1, \ldots t_n\, p_n) \in \mathtt{opns}[ct]$ resp.), $ci \in \mathtt{cinsts}[c\sigma]_{ct}$ and $v_1, \ldots, v_n$ are valid parameter values, i.e. fit to the data type parameters.
  (post) if the operation has a result type t then v is a value of data type t.

General assumption: From now on, we assume given, whenever we consider a component system signature $C\Sigma$, a fixed $C\Sigma$-model M. We do this, since we are not interested in this paper in the specification of component systems but rather on the specification of ensembles built over a given component system semantically represented by M.

Ensemble states. Let $E\Sigma = (rtypes, canPlay)$ be an ensemble signature over a component system signature $C\Sigma$ with $canPlay \subseteq C\Sigma \times rtypes$. An *ensemble state* over a component system state $c\sigma = (cinsts, cdata)$ is a triple

$$e\sigma = (rinsts, playedBy, ctrl), \text{ where}$$

$rinsts = \bigcup_{rt \in rtypes} rinsts_{rt}$ is the disjoint union of finite sets $rinsts_{rt}$ of (identifiers for) currently existing role instances of type rt, $playedBy : rinsts \rightarrow cinsts$ is a function assigning to each role instance $ri \in rinsts_{rt}$ a component instance $ci \in cinsts_{ct}$ with $(ct, rt) \in canPlay$, and *ctrl* is a global control state recording the current execution state of an ensemble.

We write $\mathtt{rinsts}[e\sigma]$ for *rinsts*, $\mathtt{playedBy}[e\sigma]$ for *playedBy* and $\mathtt{ctrl}[e\sigma]$ for *ctrl*. Moreover, we write $\mathtt{playedBy}[e\sigma][rj \mapsto cj]$ for the updated function with

[2] It is not important here whether the transition relation is deterministic or not, but if we think on standard implementations of operations, like in object-oriented languages, the execution of an operation is deterministic.

$\texttt{playedBy}[e\sigma][rj \mapsto cj](rj) = cj$ and $\texttt{playedBy}[e\sigma][rj \mapsto cj](ri) = \texttt{playedBy}[e\sigma](ri)$ for $ri \neq rj$. The set of pairs $(e\sigma, c\sigma)$ where $e\sigma$ is an ensemble state over component state $c\sigma$ is denoted by $States(E\Sigma, C\Sigma)$.

Labels. Three kinds of labels are used on transitions which interpret the syntactic actions (a) - (c) defined in Sect. 2.

(a) A create label $rj = ri.\mathbf{create}(rt, cj)$ expresses that role instance ri creates a role instance rj of type rt which is then played by component instance cj.

(b) A communication label $(ri \to rj).mtnm(v_1, \ldots, v_n)$ expresses that role instance ri sends a message with name $mtnm$ to role instance rj transmitting the values $v_1, \ldots, v_n$ which are (identifiers of) role instances or data values.

(c) A component access label $ci.opnm(v_1, \ldots, v_n)$ or $v = ci.opnm(v_1, \ldots, v_n)$ is defined as before for labels of $C\Sigma$-models.

The set of labels over $E\Sigma$ and $C\Sigma$ is denoted by $Lab(E\Sigma, C\Sigma)$.

Ensemble transition systems constrain the use of labels on transitions by appropriate pre- and postconditions, like "a role instance rj of type rt can only be created on top of a component instance cj of type ct if the ensemble signature allows that roles of type rt can be played by components of type ct".

Definition 3 (Ensemble transition system). *Let $C\Sigma$ be a component system signature with model $M = (States(C\Sigma), c\sigma_0, Lab(C\Sigma), \to_M)$. Let $E\Sigma$ be an ensemble signature over $C\Sigma$. An* ensemble transition system *(shortly ETS) for $E\Sigma$ and $C\Sigma$ is a tuple $T = (States(E\Sigma, C\Sigma), (e\sigma_0, c\sigma_0), Lab(E\Sigma, C\Sigma), \to)$ such that*

- *$(e\sigma_0, c\sigma_0) \in States(E\Sigma, C\Sigma)$ is the initial ensemble state $e\sigma_0$ over $c\sigma_0$,*
- *$\to \subseteq States(E\Sigma, C\Sigma) \times Lab(E\Sigma, C\Sigma) \times States(E\Sigma, C\Sigma)$ is a transition relation such that for all $(e\sigma, c\sigma) \xrightarrow{l} (e\sigma', c\sigma')$ the following well-formedness conditions are satisfied:*
  - *(a) if l is of the form $rj = ri.\mathbf{create}(rt, cj)$ then*
    - *(pre) $ri \in \texttt{rinsts}[e\sigma], rj \notin \texttt{rinsts}[e\sigma]$, and there exists $ct \in C\Sigma$ such that $cj \in \texttt{cinsts}[c\sigma]_{ct}$ and $(ct, rt) \in \texttt{canPlay}[E\Sigma]$,*
    - *(post) $\texttt{rinsts}[e\sigma'] = \texttt{rinsts}[e\sigma] \cup \{rj\}, rj \in \texttt{rinsts}[e\sigma']_{rt}$, $\texttt{playedBy}[e\sigma'] = \texttt{playedBy}[e\sigma][rj \mapsto cj]$, and $c\sigma' = c\sigma$.*
  - *(b) if l is of the form $(ri \to rj).mtnm(v_1, \ldots, v_n)$ then*
    - *(pre) there exist $rt, rt' \in \texttt{rtypes}[E\Sigma]$ such that $mtnm(t_1\, p_1, \ldots t_n\, p_n) \in \texttt{mts}_{out}[rt] \cap \texttt{mts}_{in}[rt']$, $ri \in \texttt{rinsts}[e\sigma]_{rt}, rj \in \texttt{rinsts}[e\sigma]_{rt'}$ and $v_1, \ldots, v_n$ are valid parameter values; in particular, if a parameter type t_k is a role type then $v_k \in \texttt{rinsts}[e\sigma]_{t_k}$,*
    - *(post) $\texttt{rinsts}[e\sigma'] = \texttt{rinsts}[e\sigma], \texttt{playedBy}[e\sigma'] = \texttt{playedBy}[e\sigma]$, and $c\sigma' = c\sigma$.*
  - *(c) if l is of the form $ci.opnm(v_1, \ldots, v_n)$ ($v = ci.opnm(v_1, \ldots, v_n)$ resp.) then $c\sigma \xrightarrow{ci.opnm(v_1,\ldots,v_n)}_M c\sigma'$ ($c\sigma \xrightarrow{v=ci.opnm(v_1,\ldots,v_n)}_M c\sigma'$ resp.) and, $\texttt{rinsts}[e\sigma'] = \texttt{rinsts}[e\sigma], \texttt{playedBy}[e\sigma'] = \texttt{playedBy}[e\sigma']$.*

Note that in all transitions the control state of an ensemble may change in accordance with the execution progress of the ensemble. The class of all ensemble transition systems for $E\Sigma$ and $C\Sigma$ is denoted by $Trans(E\Sigma, C\Sigma)$.

At next we define the satisfaction relation between ensemble transition systems and ensemble formulas. For this purpose, we have to consider environments ρ which map variables to values, more precisely, role instance variables to role instance identifiers and data variables to data values. The set of all environments over $E\Sigma$ is denoted by $Env(E\Sigma)$. Updating an environment ρ with a value v for a variable var is denoted by $\rho[var \mapsto v]$. We assume given an interpretation function $I_{\sigma,\rho}$ which maps, depending on a state $\sigma \in States(E\Sigma, C\Sigma)$ and an environment $\rho \in Env(E\Sigma)$, expressions to values. We do not detail the interpretation function but assume that it is inductively defined as usual along the structure of expressions. In particular, we assume that $I_{\sigma,\rho}(r.\mathbf{playedBy}) = \texttt{playedBy}[e\sigma](\rho(r))$ for any $\sigma = (e\sigma, c\sigma) \in States(E\Sigma, C\Sigma), \rho \in Env(E\Sigma)$ and $r \in RVar$.

To define the satisfaction relation for formulas of the form $\langle\alpha\rangle\varphi$ with $\alpha \in Act(E\Sigma, C\Sigma)$ we lift the semantic transition relation $\rightarrow$ of an ETS T to environments and use the syntactic actions in $Act(E\Sigma, C\Sigma)$ on the transitions. Each ensemble transition system $T = (States(E\Sigma, C\Sigma), (e\sigma_0, c\sigma_0), Lab(E\Sigma, C\Sigma), \rightarrow)$ gives rise to a transition relation

$$\twoheadrightarrow \subseteq (States(E\Sigma, C\Sigma) \times Env(E\Sigma)) \times Act(E\Sigma, C\Sigma) \times (States(E\Sigma, C\Sigma) \times Env(E\Sigma))$$

which is constructed according to the rules in Fig. 2.

The first four rules have transitions of T (denoted by $\rightarrow$) in their premises. The fourth rule considers component access for pure operations. The case of operations with result types is analogous but needs two rules similarly to the two create rules. The other rules deal with composed, structured actions and have transitions of the form $\twoheadrightarrow$ in their premises.

Then, for any state $\sigma \in States(E\Sigma, C\Sigma)$ and environment $\rho \in Env(E\Sigma, C\Sigma)$, the satisfaction of ensemble formulas by T is inductively defined as follows:

- $T, \sigma, \rho \models \mathbf{tt}$,
- $T, \sigma, \rho \models r = p$ if $\rho(r) = \rho(p)$,
- $T, \sigma, \rho \models \neg\varphi$ if not $T, \sigma, \rho \models \varphi$,
- $T, \sigma, \rho \models \varphi \vee \psi$ if $T, \sigma, \rho \models \varphi$ or $T, \sigma, \rho \models \psi$,
- $T, \sigma, \rho \models \langle\alpha\rangle\varphi$ if there exist $(\sigma', \rho') \in States(E\Sigma, C\Sigma) \times Env(E\Sigma, C\Sigma)$ such that $(\sigma, \rho) \overset{\alpha}{\twoheadrightarrow} (\sigma', \rho')$ and $T, \sigma', \rho' \models \varphi$,
- $T, \sigma, \rho \models \exists r{:}rt.\varphi$ if there exists $ri \in \texttt{rinsts}[\sigma]_{rt}$ such that $T, \sigma, \rho[r \mapsto ri] \models \varphi$.

If φ is a sentence the environment ρ is irrelevant. T *satisfies* a sentence $\varphi \in \mathrm{Sen}(E\Sigma, C\Sigma)$, denoted by $T \models \varphi$, if $T, \sigma_0 \models \varphi$ with $\sigma_0 = (e\sigma_0, c\sigma_0)$.

(create1)
$$\frac{\sigma \xrightarrow{rj=ri.\mathbf{create}(cj)} \sigma'}{(\sigma,\rho) \xrightarrow{p:=r.\mathbf{create}(rt,ce)}\!\!\!\!\rightarrow (\sigma', \rho[p \mapsto rj])}$$
whenever $p \neq \mathbf{any}$ and $(r = \mathbf{any}$ or $\rho(r) = ri)$ and $(ce = \mathbf{any}$ or $I_{\sigma,\rho}(ce) = cj)$

(create2)
$$\frac{\sigma \xrightarrow{rj=ri.\mathbf{create}(cj)} \sigma'}{(\sigma,\rho) \xrightarrow{\mathbf{any}:=r.\mathbf{create}(rt,ce)}\!\!\!\!\rightarrow (\sigma', \rho)}$$
whenever $(r = \mathbf{any}$ or $\rho(r) = ri)$ and $(ce = \mathbf{any}$ or $I_{\sigma,\rho}(ce) = cj)$

(comm)
$$\frac{\sigma \xrightarrow{(ri\to rj).mtnm(v_1,\ldots,v_n)} \sigma'}{(\sigma,\rho) \xrightarrow{(r\to p).mtnm(e_1,\ldots,e_n)}\!\!\!\!\rightarrow (\sigma', \rho)}$$
whenever $(r = \mathbf{any}$ or $\rho(r) = ri)$ and $(p = \mathbf{any}$ or $\rho(r) = rj)$ and, for $i = 1, \ldots, n$, $(e_i = \mathbf{any}$ or $I_{\sigma,\rho}(e_i) = v_i)$

(comp access)
$$\frac{\sigma \xrightarrow{ci.opnm(v_1,\ldots,v_n)} \sigma'}{(\sigma,\rho) \xrightarrow{r.\mathbf{playedBy}.opnm(e_1,\ldots,e_n)}\!\!\!\!\rightarrow (\sigma', \rho)}$$
whenever $(r = \mathbf{any}$ or $I_{\sigma,\rho}(r.\mathbf{playedBy}) = ci)$ and, for $i = 1, \ldots, n$, $(e_i = \mathbf{any}$ or $I_{\sigma,\rho}(e_i) = v_i)$

(assignment)
$$(\sigma,\rho) \xrightarrow{r:=p}\!\!\!\!\rightarrow (\sigma, \rho[r \mapsto \rho(p)])$$
for all $(\sigma,\rho) \in States(E\Sigma, C\Sigma) \times Env(E\Sigma)$

(seq. composition)
$$\frac{(\sigma,\rho) \xrightarrow{\alpha}\!\!\!\!\rightarrow (\hat{\sigma},\hat{\rho}), (\hat{\sigma},\hat{\rho}) \xrightarrow{\beta}\!\!\!\!\rightarrow (\sigma',\rho')}{(\sigma,\rho) \xrightarrow{\alpha;\beta}\!\!\!\!\rightarrow (\sigma',\rho')}$$

(union)
$$\frac{(\sigma,\rho) \xrightarrow{\alpha}\!\!\!\!\rightarrow (\sigma',\rho')}{(\sigma,\rho) \xrightarrow{\alpha+\beta}\!\!\!\!\rightarrow (\sigma',\rho')} \qquad \frac{(\sigma,\rho) \xrightarrow{\beta}\!\!\!\!\rightarrow (\sigma',\rho')}{(\sigma,\rho) \xrightarrow{\alpha+\beta}\!\!\!\!\rightarrow (\sigma',\rho')}$$

(iteration refl.)
$$(\sigma,\rho) \xrightarrow{\alpha^*}\!\!\!\!\rightarrow (\sigma,\rho)$$
for all $(\sigma,\rho) \in States(E\Sigma, C\Sigma) \times Env(E\Sigma)$

(iteration trans.)
$$\frac{(\sigma,\rho) \xrightarrow{\alpha^*}\!\!\!\!\rightarrow (\hat{\sigma},\hat{\rho}), (\hat{\sigma},\hat{\rho}) \xrightarrow{\alpha}\!\!\!\!\rightarrow (\sigma',\rho')}{(\sigma,\rho) \xrightarrow{\alpha^*}\!\!\!\!\rightarrow (\sigma',\rho')}$$

Fig. 2: Lifting from semantic labels to syntactic actions and environments

Definition 4 (Semantics of ensemble specifications and refinement). *Let $C\Sigma$ be a component system signature and $EnsSpec = (E\Sigma, \Phi, \phi_0)$ be an ensemble specification over $C\Sigma$. A* model *of EnsSpec is an ETS which satisfies Φ and ϕ_0. The* semantics *of EnsSpec is given by its model class, i.e. by the class*

$$Mod(EnsSpec) = \{T \in Trans(E\Sigma, C\Sigma) \mid T \models \varphi \text{ for all } \varphi \in \Phi \cup \{\phi_0\}\}.$$

An ensemble specification $EnsSpec' = (E\Sigma, \Phi', \phi_0')$ is a refinement *of EnsSpec if $\emptyset \neq Mod(EnsSpec') \subseteq Mod(EnsSpec)$.*

As an equivalence relation for ETSs we use ensemble bisimulation. In contrast to the usual bisimulation relation between processes, special care must be taken about the treatment of role instances. We abstract from the particular names of role instances by using, for related ensemble states $e\sigma_1 = (rinsts_1, playedBy_1, c_1)$ and $e\sigma_2 = (rinsts_2, playedBy_2, c_2)$, a *role instance mapping* between $rinsts_1$ and $rinsts_2$. A role instance mapping is a bijective function $\kappa : rinsts_1 \to rinsts_2$ which (a) preserves role types and (b) is compatible with the "playedBy" functions. This means that (a) $\kappa((rinsts_1)_{rt}) = (rinsts_2)_{rt}$ for all role types rt of the ensemble signature, and (b) $playedBy_1(ri) = playedBy_2(\kappa(ri))$ for each $ri \in rinsts_1$. Note, however, that there may be many role instances in $rinsts_1$ and $rinsts_2$ which are played by the same component instance, since the "playedBy" functions are usually neither injective nor surjective. We assume above that $e\sigma_1$ and $e\sigma_2$ are ensemble states over the same component system state $c\sigma$. As a consequence of this discussion, our bisimulation relation is ternary and relates states in accordance with a bijective mapping between role instances.

Remark 1. One may wonder whether role instance mappings κ must really be bijective functions or whether the use of relations would be sufficient. A first observation shows that satisfaction of sentences involving equations $r = p$ (with role instance variables r and p) would, in general, not be preserved and reflected by ensemble bisimulation if κ is not a bijective function. So, let us consider for a moment a sub-logic without equations. In order to preserve and reflect satisfaction of sentences of the form $\exists r{:}rt.\varphi$, κ must at least be a relation which is surjective in both directions. We suggest that then satisfaction of sentences would be invariant under ensemble bisimulation. On the other hand, we claim that - besides very simple examples - an ensemble bisimulation can anyway only be established in terms of a bijective function between role instances.

Definition 5 (Ensemble bisimulation). *Let $C\Sigma$ be a component system signature and $E\Sigma$ be an ensemble signature over $C\Sigma$. Let*
$T_1 = (States(E\Sigma, C\Sigma), (e\sigma_{1,0}, c\sigma_0), Lab(E\Sigma, C\Sigma), \to_1)$ *and*
$T_2 = (States(E\Sigma, C\Sigma), (e\sigma_{2,0}, c\sigma_0), Lab(E\Sigma, C\Sigma), \to_2)$
be two ensemble transition systems for $E\Sigma$ and $C\Sigma$.
Let $\Delta = \{(\sigma_1, \sigma_2, \kappa) \mid \sigma_1 = (e\sigma_1, c\sigma), \sigma_2 = (e\sigma_2, c\sigma) \in States(E\Sigma, C\Sigma)$, $\kappa : \texttt{rinsts}[e\sigma_1] \to \texttt{rinsts}[e\sigma_2]$ is a role instance mapping$\}$.
A bisimulation relation between T_1 *and* T_2 *is a relation $R \subseteq \Delta$, such that for all $(\sigma_1, \sigma_2, \kappa) \in R$ the following holds:*

(1.1) If $\sigma_1 \xrightarrow{rj_1 = ri_1.\textbf{create}(rt,cj)}_1 \sigma_1'$ *then there exist* σ_2' *and* $\sigma_2 \xrightarrow{rj_2 = \kappa(ri_1).\textbf{create}(rt,cj)}_2 \sigma_2'$ *such that* $(\sigma_1', \sigma_2', \kappa[rj_1 \mapsto rj_2]) \in R$.

(1.2) If $\sigma_1 \xrightarrow{(ri_1 \to rj_1).mtnm(v_1,\ldots,v_n)}_1 \sigma_1'$ *then there exist* σ_2' *and* $\sigma_2 \xrightarrow{(\kappa(ri_1) \to \kappa(rj_1)).mtnm(\overrightarrow{\kappa}(v_1,\ldots,v_n))}_2 \sigma_2'$ *such that* $(\sigma_1', \sigma_2', \kappa) \in R$. $\overrightarrow{\kappa}(v_1, \ldots, v_n)$ *denotes the pointwise application of* κ *to those values which are role instances.*

(1.3) If $\sigma_1 \xrightarrow{v = ci.opnm(v_1,\ldots,v_n)}_1 \sigma_1'$ *then there exist* σ_2' *and* $\sigma_2 \xrightarrow{v = ci.opnm(v_1,\ldots,v_n)}_2 \sigma_2'$ *such that* $(\sigma_1', \sigma_2', \kappa) \in R$.

(2.1) If $\sigma_2 \xrightarrow{rj_2 = ri_2.\textbf{create}(rt,cj)}_2 \sigma_2'$ *then there exist* σ_1' *and* $\sigma_1 \xrightarrow{rj_1 = \kappa^{-1}(ri_2).\textbf{create}(rt,cj)}_1 \sigma_1'$ *such that* $(\sigma_1', \sigma_2', \kappa[rj_1 \mapsto rj_2]) \in R$.

(2.2) If $\sigma_2 \xrightarrow{(ri_2 \to rj_2).mtnm(v_1,\ldots,v_n)}_2 \sigma_2'$ *then there exist* σ_1' *and* $\sigma_1 \xrightarrow{(\kappa^{-1}(ri_2) \to \kappa^{-1}(rj_2)).mtnm(\overrightarrow{\kappa^{-1}}(v_1,\ldots,v_n))}_1 \sigma_1'$ *such that* $(\sigma_1', \sigma_2', \kappa) \in R$.

(2.3) If $\sigma_2 \xrightarrow{v = ci.opnm(v_1,\ldots,v_n)}_2 \sigma_2'$ *then there exist* σ_1' *and* $\sigma_1 \xrightarrow{v = ci.opnm(v_1,\ldots,v_n)}_2 \sigma_2'$ *such that* $(\sigma_1', \sigma_2', \kappa) \in R$.

The rules (1.3) and (2.3) consider component access for operations with results. The case of pure operations is analogously and omitted here.

T_1 *and* T_2 *are* bisimulation equivalent*, denoted by* $T_1 \sim_e T_2$*, if there exists a bisimulation relation* $R \subseteq \Delta$ *between* T_1 *and* T_2 *and* $\kappa_0 : \texttt{rinsts}[e\sigma_{1,0}] \to \texttt{rinsts}[e\sigma_{2,0}]$ *such that* $((e\sigma_{1,0}, c\sigma_0), (e\sigma_{2,0}, c\sigma_0), \kappa_0) \in R$.

The following theorem, part (1), shows that the satisfaction of sentences is invariant (i.e. preserved and reflected) under ensemble bisimulation. The proof relies on the fact that any bisimulation relation between two ETSs T_1 and T_2 can be lifted from semantic labels to syntactic labels, i.e. actions in $Act(E\Sigma, C\Sigma)$. Part (2) of the theorem shows, that, if the two ETSs are image-finite[3] then also the converse holds. Thus the modal logic for ensembles satisfies the Hennessy-Milner property. Theorem 1 extends an analogous theorem in [9] by taking into account components and component access.

Theorem 1. *Let* T_1 *and* T_2 *be two ETSs for an ensemble signature* $E\Sigma$ *and component system signature* $C\Sigma$*.*

(1) If $T_1 \sim_e T_2$ *then, for any sentence* $\varphi \in \mathrm{Sen}(E\Sigma, C\Sigma)$*,* $T_1 \models \varphi$ *iff* $T_2 \models \varphi$*.*
(2) If T_1 *and* T_2 *are image-finite then the converse of* (1) *holds, i.e. if for any* $\varphi \in \mathrm{Sen}(E\Sigma, C\Sigma)$*,* $T_1 \models \varphi$ *iff* $T_2 \models \varphi$*, then* $T_1 \sim_e T_2$*.*

As a consequence of (1), the model class $Mod(EnsSpec)$ of an ensemble specification *EnsSpec* is closed under bisimulation equivalence.

[3] This means that in any state there are at most finitely many outgoing transitions labelled with the same action. In particular, for any create action (see Definition 3(a)), the instance *rj* should be chosen from a finite set of instance identifiers.

4 Ensemble Realisations

Ensemble specifications describe properties of collaborating ensemble participants from a gobal perspective. In this section we consider ensemble realisations which define, for each role type rt of an ensemble signature, a local behaviour to be respected by all instances of rt. Behaviours are described in a constructive way by process expressions. Given a component signature $C\Sigma$ and an ensemble signature $E\Sigma = (rtypes, canPlay)$ over $C\Sigma$, process expressions and (local) role actions are defined by the following grammar:[4]

$$
\begin{aligned}
P &::= \mathbf{nil} \mid a.P \mid P_1 + P_2 \mid rt \\
a &::= p := \mathbf{create}(rt, \mathbf{playedBy}) \mid p := \mathbf{create}(rt, \mathbf{playedBy}.att) \mid \\
&\quad ?mtnm(t_1\, p_1, \ldots, t_n\, p_n) \mid !p.mtnm(e_1, \ldots, e_n) \mid \\
&\quad [t\, d :=]\mathbf{playedBy}.opnm(e_1, \ldots, e_n)
\end{aligned}
$$

In this grammar rt ranges over the role types in $rtypes$, p is a process instance variable, att is a reference attribute, $mtnm$ ranges over the names of message types in $\mathtt{mts}[E\Sigma]$, $t_1\, p_1, \ldots, t_n\, p_n$ are formal parameters, $e_1, \ldots, e_n$ are expressions, $opnm$ ranges over the names of operations in $\mathtt{opns}[C\Sigma]$, and d is a variable of type t (if t is the result type of $opnm$). We assume that there is a predefined variable **self** which can be used as an actual role type parameter in messages in order to transmit the identity of a role instance for possible callbacks.

The set of process expressions over $E\Sigma$ and $C\Sigma$ is denoted by $PExp(E\Sigma, C\Sigma)$. **nil** denotes the null process, $a.P$ action prefix, $P_1 + P_2$ non-deterministic choice and rt process invocation. In contrast to communication actions in $Act(E\Sigma, C\Sigma)$, process expressions contain distinguished receive and send actions seen from the perspective of a single role instance. A receive action $?mtnm(t_1\, p_1, \ldots t_n\, p_n)$ expresses that the current role instance is enabled to receive a message with values $v_1, \ldots, v_n$ of types $t_1, \ldots, t_n$ which will be stored in local variables $p_1, \ldots, p_n$ of the instance. A send action $!p.mtnm(e_1, \ldots, e_n)$ expresses that the current role instance is enabled to send a message to the role instance, denoted by the local variable p, transmitting the values of the expressions $e_1, \ldots, e_n$. A create action $p := \mathbf{create}(rt, \mathbf{playedBy})$ expresses that the current role instance is enabled to create a role instance of type rt on its owning component instance or, if the parameter is $\mathbf{playedBy}.att$, on the component instance accessed by the reference attribute att. The new role instance is stored in the local variable p. A component access action $\mathbf{playedBy}.opnm(e_1, \ldots, e_n)$ expresses that the current role instance is enabled to call the operation $opnm$ on its owning component instance handing over the values of the expressions $e_1, \ldots, e_n$. If the operation delivers a result then the result is stored in the local variable d of the role instance.

Definition 6 (Ensemble realisation). *Let $C\Sigma$ be a component system signature with model M. An* ensemble realisation *over $C\Sigma$ is a triple $EnsReal = (E\Sigma, Reals, (rinsts_0, playedBy_0))$ where $E\Sigma = (rtypes, canPlay)$ is an ensemble*

[4] A more expressive syntax allowing, e.g., conditional expressions with Boolean guards can be found in [12].

signature over $C\Sigma$*,* $Reals = \{rt = P_{rt} \mid rt \in rtypes\}$ *is a set of* role type realisations *with* $P_{rt} \in PExp(E\Sigma, C\Sigma)$*,* $rinsts_0$ *is a non-empty set of initially existing role instances and* $playedBy_0 : rinsts_0 \to \texttt{cinsts}[c\sigma_0]$ *assigns to each* $ri \in rinsts_0$ *a component instance of the initial state* $c\sigma_0$ *of* M*.*

The semantics of an ensemble realisation is given in terms of an ensemble transition system. In this case the global control state `ctrl` of an ensemble state $e\sigma = (rinsts, playedBy, ctrl)$ has a particular form: it is a function $ctrl : rinsts \to LStates(E\Sigma)$ assigning to each currently existing role instance $ri \in rinsts$ a local state. A *local state* is a pair $l = (\eta, P)$ where η is valuation of the local variables of role instance ri and P is a process expression recording the current computation state of ri. We write $\texttt{val}[l]$ for η and $\texttt{proc}[l]$ for P. The set of all local states over $E\Sigma$ is denoted by $LStates(E\Sigma)$. The set of all local variable valuations is denoted by $Val(E\Sigma)$. Updating a valuation η with a value v for a variable var is denoted by $\eta[var \mapsto v]$. The valuation of an empty set of local variables is denoted by $\emptyset$. Similarly to Section 3 we assume given an interpretation function $I_{\sigma,\eta}$ which maps, depending on a state $\sigma \in States(E\Sigma, C\Sigma)$ and a local variable valuation $\eta \in Val(E\Sigma)$, expressions to values.

An ensemble realisation $EnsReal = (E\Sigma, Reals, (rinsts_0, playedBy_0))$ determines the set $rinsts_0$ of role instances when the ensemble starts its execution. This set determines also a starting control state $ctrl_0$ of the ensemble which maps, for all role types rt of $E\Sigma$, each $ri \in (rinsts_0)_{rt}$ to the local state $(\emptyset[\mathbf{self} \mapsto ri], P_{rt})$ where $rt = P_{rt}$ is the realisation of rt in $Reals$. In summary we obtain the *initial ensemble realisation state* $e\sigma_0 = (rinsts_0, playedBy_0, ctrl_0)$.

In contrast to the loose semantics of ensemble specifications, an ensemble realisation determines, up to identifiers of newly created role instances, a unique ensemble transition system. Structural operational semantics (SOS) rules define the allowed transitions. We pursue an incremental approach, similar to the Fork Calculus in [5], by splitting the semantics into two different layers. The first layer describes how a process expression evolves according to the given constructs for process expressions. The second layer builds on the first one by defining the evolution of ensemble realisation states $e\sigma$ over component system states $c\sigma$.

Evolution of process expressions: Fig. 3 provides the SOS rules for the progress of process expressions. The rule for process invocation relies on the role type realisations $Reals$ given in an ensemble realisation. We use the symbol $\hookrightarrow$ for transitions on the process level.

Evolution of Ensembles: On the next level we consider ensemble realisation states and their transitions denoted by $\to$ in Fig. 4. The transitions in Fig. 4 are derived as follows: create actions $p := \mathbf{create}(rt, \mathbf{playedBy})$ on the process type level cause the creation of a new role instance in a given ensemble state $e\sigma$ which is now played by the component instance being the owner of the creating role instance. We use the notation $fresh(e\sigma, rt)$ to refer to the choice of a unique role instance of type rt, which does not belong to $\texttt{rinsts}[e\sigma]_{rt}$. We have omitted in Fig. 4 the rule for create actions of the form $p := \mathbf{create}(rt, \mathbf{playedBy}.att)$ which is analogous, but now the created role is played by the component instance

(action prefix)	$a.P \overset{a}{\hookrightarrow} P$
(choice-left)	$\dfrac{P_1 \overset{a}{\hookrightarrow} P_1'}{P_1 + P_2 \overset{a}{\hookrightarrow} P_1'}$
(choice-right)	$\dfrac{P_2 \overset{a}{\hookrightarrow} P_2'}{P_1 + P_2 \overset{a}{\hookrightarrow} P_2'}$
(process type invocation)	$\dfrac{rt = P_{rt} \in \mathit{Reals},\ P_{rt} \overset{a}{\hookrightarrow} P'}{rt \overset{a}{\hookrightarrow} P'}$

Fig. 3: SOS rules for process expressions

obtained by evaluation of the reference attribute *att*. Let us now consider communication inside an ensemble by message exchange. In the semantics presented here we use synchronous, binary communication - rule (comm) - where message output and message input are performed simultaneously when role instances are able to communicate.

Definition 7 (Semantics of an ensemble realisation). *Let $C\Sigma$ be a component system signature with model M and EnsReal be an ensemble realisation with signature $E\Sigma$ over $C\Sigma$. The semantics of EnsReal is the ensemble transition system*

$$[\![\mathit{EnsReal}]\!] = (\mathit{States}(E\Sigma, C\Sigma), (e\sigma_0, c\sigma_0), \mathit{Lab}(E\Sigma, C\Sigma), \rightarrow)$$

where $e\sigma_0$ is the initial ensemble realisation state (derived from EnsReal as explained above), $c\sigma_0$ is the initial component system state of M and $\rightarrow$ is the transition relation generated from $(e\sigma_0, c\sigma_0)$ by applying the rules in Fig. 3 and Fig. 4.

Note that the rules in Fig. 4 guarantee the constraints for an ensemble transition system formulated in Definition 3. Our semantic concepts lead to an obvious correctness notion for ensemble specifications and their realisations:

Definition 8 (Correct ensemble realisation). *Let EnsSpec be an ensemble specification and EnsReal be an ensemble realisation over the same signature. EnsReal is a* correct realisation *of EnsSpec if $[\![\mathit{EnsReal}]\!] \in \mathit{Mod}(\mathit{EnsSpec})$.*

Example 3. We provide a realisation of the file transfer ensemble which satisfies the specification in Example 2. The behaviour of each of the three role types is defined in Fig. 5. For the initial state of the component system model we assume that there exists a fixed number of component instances of type `Peer` organised in a ring structure. For the initial ensemble state we require that

(create) $$\frac{P_i \xhookrightarrow{p:=\mathbf{create}(rt,\mathbf{playedBy})} P_i'}{(e\sigma, c\sigma) \xrightarrow{fresh(e\sigma,rt)=ri.\mathbf{create}(rt,ci)} (e\sigma', c\sigma)}$$

whenever $ri \in \mathtt{rinsts}[e\sigma], \mathtt{proc}[\mathtt{ctrl}[e\sigma](ri)] = P_i$, and,
there exists $ct \in C\Sigma$ such that :
$\mathtt{playedBy}[e\sigma](ri) = ci \in \mathtt{cinsts}[c\sigma]_{ct}, (ct, rt) \in \mathtt{canPlay}[E\Sigma]$,
$\mathtt{rinsts}[e\sigma'] = \mathtt{rinsts}[e\sigma] \cup \{fresh(e\sigma, rt)\}$,
$\mathtt{playedBy}[e\sigma'] = \mathtt{playedBy}[e\sigma][fresh(e\sigma, rt) \mapsto ci]$,
and, for $\eta = \mathtt{val}[\mathtt{ctrl}[e\sigma](ri)]$,
$\mathtt{ctrl}[e\sigma'] = \mathtt{ctrl}[e\sigma][ri \mapsto (\eta[p \mapsto fresh(e\sigma, rt)], P_i')]$
$[fresh(e\sigma, rt) \mapsto (\emptyset[\mathbf{self} \mapsto fresh(e\sigma, rt)], P_{rt})]$
where $rt = P_{rt} \in Reals$.

(comm) $$\frac{P_i \xhookrightarrow{!p.mtnm(e_1,\ldots,e_n)} P_i',\ P_j \xhookrightarrow{?mtnm(t_1\,p_1,\ldots,t_n\,p_n)} P_j'}{(e\sigma, c\sigma) \xrightarrow{(ri\to rj).mtnm(v_1,\ldots,v_n)} (e\sigma', c\sigma)}$$

whenever there exist $rt, rt' \in \mathtt{rtypes}[E\Sigma]$ such that :
$mtnm(t_1\,p_1, \ldots, t_n\,p_n) \in \mathtt{mts}_{out}[rt] \cap \mathtt{mts}_{in}[rt']$,
$ri \in \mathtt{rinsts}[e\sigma]_{rt}, \mathtt{proc}[\mathtt{ctrl}[e\sigma](ri)] = P_i$,
$rj \in \mathtt{rinsts}[e\sigma]_{rt'}, \mathtt{proc}[\mathtt{ctrl}[e\sigma](rj)] = P_j, rj \neq ri$,
and, for $\eta_i = \mathtt{val}[\mathtt{ctrl}[e\sigma](ri)]$, we have $\eta_i(p) = rj$,
$I_{\sigma,\eta_i}(e_k) = v_k$ are valid parameter values for $k = 1, \ldots, n$,
$\mathtt{rinsts}[e\sigma'] = \mathtt{rinsts}[e\sigma], \mathtt{playedBy}[e\sigma'] = \mathtt{playedBy}[e\sigma]$, and
$\mathtt{ctrl}[e\sigma'] = \mathtt{ctrl}[e\sigma][ri \mapsto (\eta_i, P_i')][rj \mapsto (\eta_j[p_1 \mapsto v_1] \ldots [p_n \mapsto v_n], P_j')]$
where $\eta_j = \mathtt{val}[\mathtt{ctrl}[e\sigma](rj)]$.

(comp) $$\frac{P_i \xhookrightarrow{t\,d:=\mathbf{playedBy}.opnm(e_1,\ldots,e_n)} P_i',}{(e\sigma, c\sigma) \xrightarrow{v=\mathtt{playedBy}[e\sigma](ri).opnm(v_1,\ldots,v_n)} (e\sigma', c\sigma')}$$

whenever $ri \in \mathtt{rinsts}[e\sigma], \mathtt{proc}[\mathtt{ctrl}[e\sigma](ri)] = P_i$,
$\mathtt{rinsts}[e\sigma'] = \mathtt{rinsts}[e\sigma], \mathtt{playedBy}[e\sigma'] = \mathtt{playedBy}[e\sigma]$,
and, for $\eta = \mathtt{val}[\mathtt{ctrl}[e\sigma](ri)]$,
$\mathtt{ctrl}[e\sigma'] = \mathtt{ctrl}[e\sigma][ri \mapsto (\eta[d \mapsto v], P_i')]$, and
$c\sigma \xrightarrow{v=\mathtt{playedBy}[e\sigma](ri).opnm(v_1,\ldots,v_n)}_M c\sigma'$.

Fig. 4: SOS rules for ensembles

there exists exactly one role instance which is of type `Requester` and played by the first peer. Let us remark that the role behaviour specified for a router is non-deterministic (with internal choice) for what concerns the creation of a new router or a new provider role instance. It does not take into account when the owning component of a router stores the requested file and therefore will adapt the role of a provider. This could, however, be easily done if we would use the full HELENA language [12] which includes guarded choice. Moreover, as already discussed earlier, the name of the requested file is fixed which could be avoided if we would use open systems where the requester first would expect an input of a file name from the environment. □

```
roleBehaviour Requester =
  router :=
      create(Router,playedBy.neighbour) .
  !router.reqAddr(self,''song'') .
  ((  ?sndAddr(Provider prov) .
      !prov.reqFile(self,''song'') .
      ?sndFile(File f) . nil)
   +
   (  ?notFound() . nil))

roleBehaviour Provider =
  ?reqFile(Requester req, String s) .
  File song = playedBy.getFile(''song'').
  !req.sndFile(song) . nil
```

```
roleBehaviour Router =
  ?reqAddr(Requester req, String s) .
  ( router :=
      create(Router,playedBy.neighbour).
    !router.reqAddr(req,''song'') . nil)
  +
  ( prov := create(Provider,playedBy) .
    !req.sndAddr(prov) . nil)
  +
  ( !req.notFound() . nil)
```

Fig. 5: Realisation of the file transfer ensemble

As an immediate consequence of Theorem 1(1) we obtain:

Theorem 2 (Equivalent correct ensemble realisations). *Let EnsSpec be an ensemble specification and* $EnsReal_1$ *and* $EnsReal_2$ *be two equivalent ensemble realisations, i.e.,* $[\![EnsReal_1]\!] \sim_e [\![EnsReal_2]\!]$*. Then* $EnsReal_1$ *is a correct realisation of EnsSpec if and only if* $EnsReal_2$ *is a correct realisation of EnsSpec.*

It remains the question how to prove that two ensemble realisations $EnsReal_1$ and $EnsReal_2$ are equivalent? The idea is to look to the role type realisations $rt = P_{1,rt}$ and $rt = P_{2,rt}$ of $EnsReal_1$ and $EnsReal_2$ and to show that, for each role type rt, the process expressions $P_{1,rt}$ and $P_{2,rt}$ are bisimulation equivalent in the usual sense of process algebra. We claim that this implies global bisimulation equivalence of their generated ensemble transition systems. A formal proof of this fact in the absence of component types has been given in [9].

5 Conclusion

In this work we have extended the (constructive) HELENA approach for modelling ensemble-based systems by a logic for specifying properties of ensembles. The

logic should be useful for any kind of distributed system where cooperation is a central requirement. It is also useful for specifying allowed and forbidden scenarios which underlie use case driven approaches to software development. Our logic complements temporal logics, as used, e.g., in [8] for the verification of HELENA models, since it focuses on interactions and scenarios. Of course, more case studies are still needed to validate the expressiveness of our logic.

Our approach relies on a rigorous discrimination between syntax and semantics which allows us to study ensemble bisimulation and a correctness notion for ensemble realisations. Currently this correctness notion relies on synchronous message passing in (the semantics of) ensemble realisations. In future work this should be extended to support also asynchronous communication styles, like in [12,8] or in asynchronous multiparty session types [10]. In contrast to our approach, the framework of multiparty session types is strongly influenced by the π-calculus. It is not aimed at a logic but at process algebraic descriptions of global interaction protocols from which realisations (in the form of sets of local types) can be extracted by projection. An approach to specifying multiparty sessions carrying a logical flavour is given by the global types in [2]. Such global types use also compound actions, like sequential composition and iteration, and, moreover, are able to specify unconstrained composition of parallel activities which would also be an issue for extension of our logic. But they rely on a fixed number of ensemble participants and do not support modalities and negation. Moreover, to the best of our knowledge, the session type formalisms do not distinguish between components and roles which is a crucial aspect of our approach.

Further aspects for future research are investigating methods for proving correctness of ensemble realisations and studying open ensembles and their composition by extending [6] to components. Currently an ensemble specification describes the behaviour of one kind of ensemble but we are also interested to incorporate possibilities for talking about different kinds of ensembles in the logic and in realisations.

Acknowledgement. I am grateful for the scientific cooperation with Martin Wirsing on ensemble-based systems and his support for the current study.

References

1. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: DEECO: an ensemble-based component system. In: Proc. of the 16th ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE'13). pp. 81–90. ACM (2013)
2. Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On global types and multi-party sessions. Logical Methods in Computer Science **8**(1) (2012). https://doi.org/10.2168/LMCS-8(1:24)2012, `https://doi.org/10.2168/LMCS-8(1:24)2012`
3. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: the SCEL language. ACM Transactions on Autonomous and Adaptive Systems (TAAS) **9**(2), 1–29 (2014)

4. Harel, D., Kozen, D., Tiuryn, J. (eds.): Dynamic Logic. MIT Press (2000)
5. Havelund, K., Larsen, K.G.: The Fork Calculus. In: Proc. of the Colloquium of Automata, Languages and Programming (ICALP'93). Lecture Notes in Computer Science, vol. 700, pp. 544–557. Springer (1993)
6. Hennicker, R.: A calculus for open ensembles and their composition. In: ISoLA (1). Lecture Notes in Computer Science, vol. 9952, pp. 570–588 (2016)
7. Hennicker, R., Klarl, A.: Foundations for Ensemble Modeling - The HELENA Approach. In: Specification, Algebra, and Software. Lecture Notes in Computer Science, vol. 8373, pp. 359–381. Springer (2014)
8. Hennicker, R., Klarl, A., Wirsing, M.: Model-checking HELENA ensembles with Spin. In: Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday. Lecture Notes in Computer Science, vol. 9200, pp. 331–360. Springer (2015)
9. Hennicker, R., Wirsing, M.: Dynamic logic for ensembles. In: Proc. 8th International Symposium, ISoLA. Lecture Notes in Computer Science (2018, to appear)
10. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proc. of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08). pp. 273–284. ACM (2008)
11. Klarl, A.: Engineering self-adaptive systems with the role-based architecture of HELENA. In: Proc. 24th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE'15. pp. 3–8. IEEE Computer Society (2015)
12. Klarl, A.: HELENA: Handling massively distributed systems with ELaborate ENsemble Architectures. Ph.D. thesis, LMU Munich, Germany (2016)
13. Klarl, A., Cichella, L., Hennicker, R.: From HELENA ensemble specifications to executable code. In: Proc. Formal Aspects of Component Software (FACS'14). Lecture Notes in Computer Science, vol. 8997, pp. 183–190. Springer (2014)
14. 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)
15. Wirsing, M., Hölzl, M.M., Koch, N., Mayer, P. (eds.): Software Engineering for Collective Autonomic Systems - The ASCENS Approach, Lecture Notes in Computer Science, vol. 8998. Springer (2015). https://doi.org/10.1007/978-3-319-16310-9, http://dx.doi.org/10.1007/978-3-319-16310-9
16. Wirsing, M., Hölzl, M.M., Tribastone, M., Zambonelli, F.: ASCENS: engineering autonomic service-component ensembles. In: Formal Methods for Components and Objects, 10th International Symposium, FMCO 2011, Revised Selected Papers. pp. 1–24 (2011). https://doi.org/10.1007/978-3-642-35887-6_1, https://doi.org/10.1007/978-3-642-35887-6_1