Model-Checking HELENA Ensembles with Spin*

Rolf Hennicker, Annabelle Klarl, and Martin Wirsing

Ludwig-Maximilians-Universität München, Germany

Dedicated to José Meseguer

Abstract. The HELENA approach allows to specify dynamically evolving ensembles of collaborating components. It is centered around the notion of roles which components can adopt in ensembles. In this paper, we focus on the early verification of HELENA models. We propose to translate HELENA specifications into PROMELA and check satisfaction of LTL properties with Spin [11]. To prove the correctness of the translation, we consider an SOS semantics of (simplified variants of) HELENA and PROMELA and establish stutter trace equivalence between them. Thus, we can guarantee that a HELENA specification and its PROMELA translation satisfy the same LTL formulae (without *next*). Our correctness proof relies on a new, general criterion for stutter trace equivalence.

1   Introduction

The HELENA approach [10] proposes to model large distributed systems of goal-oriented collaborations of components by dynamically evolving ensembles where the participating components play certain roles. By adopting a role, a component executes a role-specific behavior. The introduction of roles allows to focus on the particular tasks which components fulfill in specific collaborations and to structure the implementation of ensemble-based systems [13,14].

Ensembles always collaborate towards some global goals. Such goals are often temporal properties which we specify by linear temporal logic (LTL) formulae [17]. In this paper, we focus on the early (pre-implementation) verification of HELENA models for their intended goals. We propose to translate HELENA specifications into PROMELA and check satisfaction of LTL properties with the model-checker Spin [11]. PROMELA is well-suited as a target language since it supports dynamic creation of concurrent processes and asynchronous communication. Our contribution is as follows: Firstly, we propose a syntactic translation of a simplified variant of HELENA (called HELENALIGHT), which restricts ensemble specifications to the core concepts of our modeling approach, to a subset of PROMELA (called PROMELALIGHT), which is sufficient to express all HELENALIGHT concepts. Secondly, we prove the correctness of the translation. For

---

this purpose, we define a formal SOS semantics for HELENALIGHT and PROMELALIGHT specifications. The latter is based on the semantics for full PROMELA in [20]. On this semantic basis, we establish stutter trace equivalence between the semantics of a HELENALIGHT specification and its PROMELALIGHT translation. Then, we reuse results from the literature [1] that satisfaction of LTL formulae (without *next*) is preserved by stutter trace equivalence. As a consequence, we can verify LTL properties for a HELENALIGHT specification by model-checking its PROMELALIGHT translation with Spin. To prove stutter trace equivalence between HELENALIGHT and PROMELALIGHT, we investigate a new, general criterion that Kripke structures are stutter trace equivalent if particular stutter simulations (called $\approx$-stutter simulations) can be established in both directions.

In Sec. 2, we explain foundations on LTL and propose our criterion for stutter trace equivalence which entails $LTL_{\backslash \mathbf{X}}$ preservation. In Sec. 3, we summarize the HELENA modeling approach and present the running example of a peer-2-peer network storing files. Sec. 4 defines syntax and semantics of HELENALIGHT and Sec. 5 for PROMELALIGHT resp. In Sec. 6, we provide the formal translation from HELENALIGHT to PROMELALIGHT and, in Sec. 7, establish the desired correctness results. Sec. 8 discusses model-checking of HELENA specifications.

Personal Note. José, Rolf, and Martin know each other since the eighties where they investigated the foundations of algebraic specifications, José with Joseph Goguen in the initial algebra setting, and Martin and Rolf inspired by the CIP program development methodology from the loose and observational semantics point of view. Behavioral specifications and equivalence of algebras were already studied by José and Joseph in [8] which was a fruitful source for the thesis of Rolf. In the beginning of the nineties, Martin was looking for an appropriate semantical framework for underpinning the typically informal or semiformal object-oriented methods with a formal framework. Rewriting logic invented by José a few years earlier appeared to be the perfect tool: a simple computational logic that supports concurrent computation, logical deduction, and object-oriented features. Martin is still very grateful that José gave him the opportunity to present his first results on integrating formal specifications into pragmatic object-oriented development at the first WRLA in 1996 [22]. Later, José, Rolf, and Martin became all members of IFIP WG 1.3. and José and Martin worked together on the algebraic foundation and analysis of modern programming paradigms including studies of the semantic foundations of multi-paradigm languages [2], the analysis of denial of service attacks [6], and the specification and correct implementation of the distributed programming language KLAIM [7]. Our current paper follows the same aim; we present the formal foundation of HELENA ensemble specifications and use results related to José's seminal paper on algebraic simulations [19] for proving the correctness of model-checking HELENA ensembles with Spin.

It is a great pleasure to know and work with José since so many years; we admire his broad and deep knowledge in many areas; it is inspiring to discuss with him and he is a very kind and warm hearted colleague and friend. We are looking forward to many further exciting scientific exchanges with José.

2 Foundations on $LTL_{\setminus \mathbf{X}}$ Preservation

In this section, we review Kripke structures, linear temporal logic (LTL), and satisfaction of LTL formulae. We propose how to induce Kripke structures from labeled transitions systems (which will be used for the semantics of HELENALIGHT and PROMELALIGHT specifications). Furthermore, we propose a criterion for stutter trace equivalence of Kripke structures which entails $LTL_{\setminus \mathbf{X}}$[1] preservation according to the literature in [1].

LTL in Kripke Structures: A Kripke structure consists of a set of states connected by (unlabeled) transitions. The states are labeled by sets of atomic propositions which hold in the state and some states are marked as initial states.

Definition 1 (Kripke Structure). *Let AP be a set of atomic propositions. A* Kripke structure *K over AP is a tuple $(S_K, I_K, \rightarrow_K, F_K)$ such that S_K is a set of states, $I_K \subseteq S_K$ is a set of initial states, $\rightarrow_K \subseteq S_K \times S_K$ is an (unlabeled) transition relation without terminal states (i.e., $\forall s \in S_K.\exists s' \in S_K \ . \ s \rightarrow_K s'$), and $F_K : S_K \rightarrow 2^{AP}$ is a labeling function associating to each state the set of atomic propositions that hold in it.*

For a Kripke structure $K = (S_K, I_K, \rightarrow_K, F_K)$, we further define: A *path of* K is an infinite sequence $p = s_0 s_1 s_2 \dots$ (with $s_i \in S_K$ for all $i \in \mathbb{N}$) such that $s_0 \in I_K$ and $s_i \rightarrow_K s_{i+1}$. A *trace of* K is an infinite sequence $t = t_0 t_1 t_2 \dots$ such that there exists a path $p = s_0 s_1 s_2 \dots$ in K and $t_i = F_K(s_i)$ for all $i \in \mathbb{N}$.

To describe temporal properties, we use linear temporal logic (LTL).

Definition 2 (LTL [1]). *Let AP be a set of atomic propositions.* LTL *formulae over AP are inductively defined by:*

$$
\begin{array}{lll}
\phi & = p \in AP & \textit{(atomic proposition)} \\
& \mid \neg \phi \mid \phi \wedge \psi & \textit{(proposition logic operators)} \\
& \mid \boldsymbol{X}\phi \mid \Diamond\phi \mid \Box\phi \mid \phi \, \boldsymbol{U} \psi & \textit{(linear temporal logic operators)}
\end{array}
$$

Disjunction, implication, and equivalence are given by the usual abbreviations. The set of LTL formulae over AP is denoted by $LTL(AP)$.

Satisfaction of LTL formulae is defined by the usual inductive definition [1].

Definition 3 (Satisfaction of LTL in Kripke Structures). *Let $K = (S_K, I_K, \rightarrow_K, F_K)$ be a Kripke structure over AP, $t = t_0 t_1 t_2 \dots$ a trace of K, and $\phi \in LTL(AP)$; $t|_i$ denotes the subsequence $t_i t_{i+1} t_{i+2} \dots$ of t.*
The satisfaction of ϕ for trace t, written $t \models \phi$, is inductively defined by

- $t \models p$, if $p \in t_0$,
- $t \models \neg \phi$, if $t \not\models \phi$,
- $t \models \phi \wedge \psi$, if $t \models \phi$ and $t \models \psi$,
- $t \models \boldsymbol{X}\phi$, if $t|_1 \models \phi$,
- $t \models \Diamond\phi$, if there exists $k \geq 0$ such that $t|_k \models \phi$,

---

[1] $LTL_{\setminus \mathbf{X}}$ is the fragment of LTL that does not contain the *next* operator $\mathbf{X}$.

– $t \models \Box\phi$, if for all $k \geq 0$ holds $t|_k \models \phi$,
– $t \models \phi \, \boldsymbol{U} \, \psi$, if there exists $k \geq 0$
  such that $t|_k \models \psi$ and for all $0 \leq j < k$ holds $t|_j \models \phi$,

The Kripke structure K satisfies an LTL formula ϕ, written $K \models \phi$, if all traces of K satisfy ϕ.

LTL in Labeled Transition Systems: In contrast to Kripke structures, labeled transition systems do not label states with atomic propositions, but transitions with actions.

Definition 4 (Labeled Transition System). *A labeled transition system (LTS) T is a tuple $(S_T, I_T, A_T, \rightarrow_T)$ such that S_T is a set of states, $I_T \subseteq S_T$ is a set of initial states, A_T is a set of actions such that the silent action $\tau \notin A_T$, and $\rightarrow_T \subseteq S_T \times (A_T \cup \tau) \times S_T$ is a labeled transition relation.*

For an LTS $T = (S_T, I_T, A_T, \rightarrow_T)$, we further define: a^* denotes a (possibly empty) sequence of a actions. If $w = a_1 \ldots a_n$ holds for some $n \in \mathbb{N}$ and $a_1, \ldots, a_n \in (A_T \cup \tau)$, then $s \xrightarrow{w}_T s'$ stands for $s = s'$, if $n = 0$, and $s \xrightarrow{a_1}_T s_1 \ldots s_{n-1} \xrightarrow{a_n}_T s'$ with appropriate $s_1, \ldots, s_{n-1}$ otherwise. The LTS T together with a set of atomic propositions AP and a satisfaction relation $s \models \phi$ (for $s \in S_T$ and $\phi \in LTL(AP)$) induces a Kripke structure $K(T) = (S_T, I_T, \rightarrow_T^{\bullet}, F)$. The labeled transition relation $\rightarrow_T$ is transformed into an unlabeled, total transition relation $\rightarrow_T^{\bullet}$ which forgets the actions and adds a new transition $s \rightarrow_T^{\bullet} s$ for each terminal state $s \in S_T$. The labeling function $F : S_T \rightarrow 2^{AP}$ is defined by $F(s) = \{p \in AP \mid s \models p\}$.

Definition 5 (Satisfaction of LTL in Labeled Transition Systems). *Let $T = (S_T, I_T, A_T, \rightarrow_T)$ be a labeled transition system, AP a set of atomic propositions, $s \models \phi$ a satisfaction relation for $s \in S_T$ and $\phi \in LTL(AP)$.*
T satisfies ϕ, written $T \models \phi$, if $K(T) \models \phi$, i.e., the induced Kripke structure $K(T)$ satisfies ϕ.

LTL$_{\backslash \mathbf{X}}$ Preservation: Lastly, we investigate when two Kripke structures satisfy the same set of $LTL_{\backslash \mathbf{X}}$ formulae. Therefore, we introduce the notion of stutter trace equivalence.

– Two paths of Kripke structures over the same set of atomic propositions AP are *stutter trace equivalent* if their traces only differ in the number of their stutter steps, i.e., there exist sets of atomic propositions $P_i \subseteq AP$ (with $i \in \mathbb{N}$) such that the traces of both paths have the form $P_0^+ P_1^+ P_2^+ \ldots$ where P_i^+ denotes a non-empty sequence of the same set P_i.
– Two Kripke structures K_1 and K_2 are *stutter trace equivalent* if for each path of K_1 there exists a stutter trace equivalent path of K_2 and vice versa.

To provide a criterion for stutter trace equivalence of Kripke structures, we propose the notion of a $\approx$-stutter simulation.

Definition 6 ($\approx$-Stutter Simulation). *Let $K_1 = (S_1, I_1, \rightarrow_1, F_1)$ and $K_2 = (S_2, I_2, \rightarrow_2, F_2)$ be two Kripke structures over AP. Let $\approx \; \subseteq S_1 \times S_2$ be a relation.*

A relation $\sim \; \subseteq S_1 \times S_2$ is a $\approx$-stutter simulation of K_1 by K_2 if (1) $\sim \; \subseteq \; \approx$ and (2) for all $s \in S_1, t \in S_2$ with $s \sim t$, if $s \rightarrow_1 s'$, then $s' \sim t$ or there exists $t \rightarrow_2 t_1 \rightarrow_2 \ldots \rightarrow_2 t_n \rightarrow_2 t'_1 \rightarrow_2 \ldots \rightarrow_2 t'_m \rightarrow_2 t'$ $(n, m \geq 0)$ such that $s \approx t_i$ for all $i \in \{1, \ldots, n\}$, $s' \approx t'_j$ for all $j \in \{1, \ldots, m\}$ and $s' \sim t'$.

K_1 is $\approx$-stutter simulated by K_2 if there exists a $\approx$-stutter simulation $\sim$ of K_1 by K_2 such that $s_0 \sim t_0$ for all $s_0 \in I_1, t_0 \in I_2$.

Stutter trace equivalence does not require preservation of the branching structure of the underlying Kripke structures. Therefore, interestingly, the notion of $\approx$-stutter simulations (compared to stutter bisimulations [19] preserving branching) is sufficient to provide a criterion whether two Kripke structures are stutter trace equivalent.

Theorem 1 (Stutter Trace Equivalence). *Let K_1 and K_2 be two Kripke structures over AP with states S_1, S_2 resp. Let $\approx \; \subseteq S_1 \times S_2$ be a property-preserving relation, i.e., for all $s \in S_1, t \in S_2$, if $s \approx t$, then $F_1(s) = F_2(t)$, and $\approx^{-1}$ its inverse relation. If K_1 is $\approx$-stutter simulated by K_2 and K_2 is $\approx^{-1}$-stutter simulated by K_1, then K_1 and K_2 are stutter trace equivalent.*

Proof. Since K_1 is $\approx$-stutter simulated by K_2, each path of K_1 is simulated by a corresponding path of K_2 such that on all paths, the states related by $\approx$ have the same properties. The same holds vice versa for $\approx^{-1}$. □

The question arises which LTL formulae are satisfied by two stutter trace equivalent Kripke structures. It is clear that the *next* operator **X** of temporal logic is not preserved since stutter steps are allowed. However, if we restrict our attention to the temporal logic $LTL_{\backslash \mathbf{X}}$, we can use a result of [1] which shows that all formulae of $LTL_{\backslash \mathbf{X}}$ are preserved. In practice, eliminating the *next* operator is not a great loss since interesting properties are not so much concerned with what happens in the next step as to what eventually happens [16].

Theorem 2 (LTL$_{\backslash \mathbf{X}}$ Preservation). *Let K_1 and K_1 be two stutter trace equivalent Kripke structures over AP. For any $LTL_{\backslash \mathbf{X}}$ formula ϕ over AP, we have $K_1 \models \phi \Leftrightarrow K_2 \models \phi$.*

Proof. The proof can be found in [1, pp. 534–535] (Thm. 7.92 and Cor. 7.93).

3 The HELENA Approach

The role-based modeling approach HELENA [10] provides concepts to describe systems where components team up in 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 (the roles of) other components. By switching between roles, the component changes its currently executed behavior. By adopting several roles in parallel, a

component can concurrently execute different behaviors and participate at the same time in different ensembles.

Components: *Component instances* are classified by *component types*. They are considered as carriers of basic information relevant across many ensembles. They provide basic capabilities to store data in attributes and to perform computations by operations. Additionally, they can be connected to other component instances by storing references to them.

Roles: Whenever a component instance joins an ensemble, the component adopts a role by creating a new *role instance* and assigning it to itself. The kind of roles a component is allowed to adopt is determined by *role types*. A role type defines role-specific attributes and a set of incoming and outgoing message types which are supported for interaction and collaboration between role instances.

P2P Example: We consider 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 `Routers` and the peer storing the requested file adopts the role of the `Provider`. All these roles can be adopted by components of type `Peer`. Fig. 1 shows the component type `Peer` and the role type `Requester` in a graphical representation similar to UML classes. 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; the file's content information is represented by the attribute `content`. A `Peer` is connected to its neighbor depicted by the association in Fig. 1a. The role type `Requester` indicates by the notation `Requester:{Peer}` that any component instance of type `Peer` can adopt that role. It stores whether it already has the file in its attribute `hasFile` and supports two incoming and two outgoing messages.
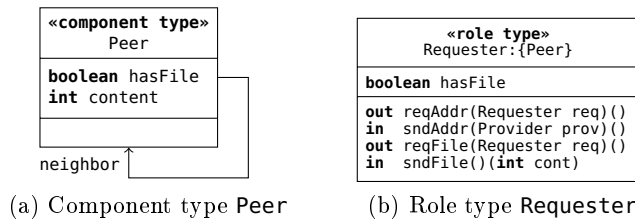

(a) Component type `Peer`   (b) Role type `Requester`

Fig. 1: Types occurring in the p2p example

Ensemble Structures: To define the structural characteristics of a collaboration, an *ensemble structure* specifies the role types whose instances form the ensemble, determines how many instances of each role type may contribute by a multiplicity (like 0..1, 1, $*$, 1..$*$ etc.), and defines the capacity of the input queue for each role type. We assume that between instances of two role types the messages which are output on one side and input on the other side can be exchanged on the input queues of the role instances.

P2P Example: Fig. 2 shows a graphical representation of the ensemble structure for the p2p example. It consists of the three role types `Requester`, `Router`, and `Provider` with associated multiplicities and input queue capacities.
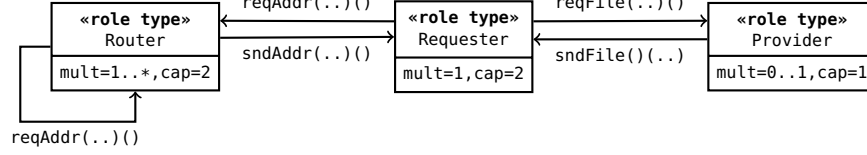
Fig. 2: Ensemble structure $\Sigma_{transfer}$ for the p2p example

Role Behaviors and Ensemble Specifications: An *ensemble specification* adds dynamic behavior to an ensemble structure Σ by equipping each role type occurring in Σ with a *role behavior*. A role behavior is given by a process expression built from the null process **nil**, action prefix $a.P$, conditional selection **if** (*condition1*) **then** $\{P1\}$(**or** (*condition2*) **then** $\{P2\}$)* (with nondeterministic choice if several branches are executable), and process invocation. There are actions for creating (**create**) and retrieving (**get**) role instances, sending (!) or receiving (?) messages, and invoking operations of the owning component. Additionally, state labels can be used to mark a certain progress of execution in the role behavior (we will use these labels in atomic propositions to express goals). We additionally use predefined variables 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 ask whether the component instance `ci` currently plays the role `rt`.

P2P Example: Fig. 3 shows the behavior specification of a `Router`. Initially, a router can receive a request for an address. Depending on whether its owner has the file, it either creates a provider role instance and sends it back to the requester in $P_{provide}$ or forwards the request to another router in P_{fwd} if possible.
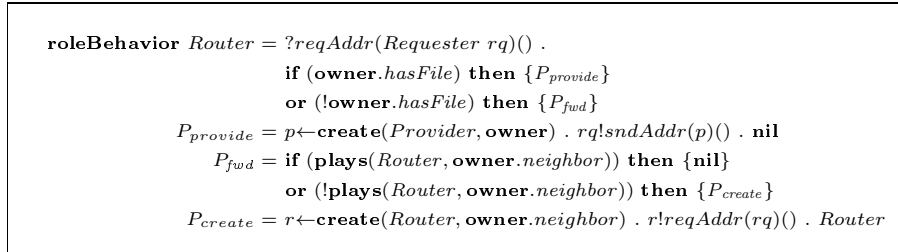
$$
\begin{aligned}
\textbf{roleBehavior } Router &= ?reqAddr(Requester\ rq)() \ . \\
&\textbf{if } (\textbf{owner}.hasFile) \textbf{ then } \{P_{provide}\} \\
&\textbf{or } (!\textbf{owner}.hasFile) \textbf{ then } \{P_{fwd}\} \\
P_{provide} &= p \leftarrow \textbf{create}(Provider, \textbf{owner}) \ . \ rq!sndAddr(p)() \ . \ \textbf{nil} \\
P_{fwd} &= \textbf{if } (\textbf{plays}(Router, \textbf{owner}.neighbor)) \textbf{ then } \{\textbf{nil}\} \\
&\textbf{or } (!\textbf{plays}(Router, \textbf{owner}.neighbor)) \textbf{ then } \{P_{create}\} \\
P_{create} &= r \leftarrow \textbf{create}(Router, \textbf{owner}.neighbor) \ . \ r!reqAddr(rq)() \ . \ Router
\end{aligned}
$$

Fig. 3: Role behavior of a `Router`

LTL for Ensemble Specifications: To express goals over HELENA ensemble specifications, we use linear temporal logic (LTL) formulae over a particular set of atomic HELENA propositions AP: A *state label proposition* is of the form $rt[i]@label$. It is satisfied if there exists a role instance i of type rt whose next performed action is the state label *label*. An *attribute proposition* must be boolean and is built from arithmetic and relational operators, data constants, and propositions of the form $rt[i]:attr$ (or $ct[i]:attr$). An attribute proposition $rt[i]:attr$ is satisfied if there exists a role instance i of type rt such that the value of its attribute *attr* evaluates to **true** (and analogously for component attributes).

7

LTL formulae over HELENA propositions and their satisfaction are inductively defined as already described in Sec. 2.

For the p2p example, we want to express that the requester will always receive the requested file if the file is available in the network. We assume a network of three peers and formulate the following achieve goal in LTL which refers to the values of the attribute `hasFile` of component type `Peer` and role type `Requester`:

$$(Peer[1]{:}hasFile \lor Peer[2]{:}hasFile \lor Peer[3]{:}hasFile) \Rightarrow \Diamond Requester[1]{:}hasFile)$$

In the next sections we will consider a simpler variant of HELENA and present a precise formalization of ensemble specifications, their semantics, satisfaction of atomic propositions and model-checking by translation to PROMELA.

4   HELENALIGHT

We restrict full HELENA specifications to some core concepts which leads to the definition of HELENALIGHT. We first formally define the syntax of HELENALIGHT ensemble specifications. Afterwards, we introduce an SOS-style semantics for such specifications and define satisfaction of LTL formulae.

4.1   Syntax of HELENALIGHT Ensemble Specifications

In HELENALIGHT, we abstract from the underlying component types of a full HELENA specification and consider only role types, whose instances can be dynamically created, and their interactions. Additionally, we omit any notion of data such that we do not consider attributes and data parameters anymore.

Role Types: Role types are characterized by their name and a set of outgoing and incoming message types. In contrast to full HELENA, we omit role attributes and consider message types with exactly one role parameter.

Definition 7 (Message Type). *A message type msg is of the form $msgnm(rt\,X)$ such that $msgnm$ is the name of the message type and X is a formal parameter of role type rt.*

Definition 8 (Role Type). *A role type is a tuple $rt = (rtnm, rtmsgs_{out}, rtmsgs_{in})$ such that $rtnm$ is the name of the role type, and $rtmsgs_{out}$ and $rtmsgs_{in}$ are sets of message types for outgoing and incoming messages supported by rt.[2]*

Ensemble Structures: Ensemble structures specify which role types are needed for a collaboration. In contrast to full HELENA, we omit multiplicities constraining the number of admissible role instances for each role type. We assume asynchronous communication and specify for each role type the (positive) capacity of the input queue of each role instance of that type.

---

[2] In the following, we often write rt synonymously for the role type name $rtnm$.

Definition 9 (Ensemble Structure). *An* ensemble structure Σ *is a tuple* $\Sigma = (nm, roletypes, roleconstraints)$ *such that nm is the name of the ensemble structure, roletypes is a set of role types, and for each* $rt \in$ *roletypes, roleconstraints(rt) is a finite capacity* $c > 0$ *of the input queue of rt.*

In this paper, we consider only closed ensemble structures Σ. This means that any outgoing message of some role type of Σ must occur as an incoming message of at least one role type of Σ and vice versa, and any parameter type occurring in a message type is a role type of Σ.

Role Behavior Declarations: Given an ensemble structure Σ, process expressions (over Σ) will be used to specify role behaviors. They are built from the process constructs and actions in Def. 10. Opposed to full HELENA, we omit component instances on which role instances are created and any data in message exchange. Furthermore, we omit the **get** action, operation calls and any attribute setters since we do not have attributes in HELENALIGHT.

Definition 10 (Process Expression). *A* process expression *is built from the following grammar, where N is the name of a process, msgnm is the name of a message type, X and Y are names of variables, rt is a role type (more precisely the name of a role type), and label is the name of a state label:*

$$
\begin{aligned}
P ::= &\ \boldsymbol{nil} & &\text{(null process)} \\
&|\ a.P & &\text{(action prefix)} \\
&|\ P_1 + P_2 & &\text{(nondeterministic choice)} \\
&|\ N & &\text{(process invocation)} \\
a ::= &\ X \leftarrow \boldsymbol{create}(rt) & &\text{(role instance creation)} \\
&|\ Y!msgnm(X) & &\text{(sending a message)} \\
&|\ ?msgnm(rt\,X) & &\text{(receiving a message)} \\
&|\ label & &\text{(state label)}
\end{aligned}
$$

A receive action $?msgnm(rt\,X)$ (and resp. a create action $X \leftarrow \boldsymbol{create}(rt)$) declares and opens the scope for a local variable X of type rt. We assume that the names of the declared variables are unique within a process expression and different from **self** which is a predefined variable that can always be used.

Definition 11 (Well-Formedness of Process Expressions). *Let* $\Sigma = (nm,$ *roletypes, roleconstraints) be an ensemble structure. A process expression* P *is* well-formed *for a role type* $rt' \in$ *roletypes w.r.t.* Σ, *if all actions occurring in* P *are well-formed for* rt' *w.r.t.* Σ. *This means:*

- *For a role instance creation action* $X \leftarrow \boldsymbol{create}(rt)$: $rt \in$ *roletypes.*

- *For a send action* $Y!msgnm(X)$,

  – *the role type* rt' *supports the message type* $msgnm(rt''\,X'')$ *as outgoing message and the variable* X *is of type* rt'',[3]

---

[3] We must distinguish here between the role type rt', whose behavior is going to be defined, and the role type rt'' used for the parameter.

- *the role type of the variable Y supports the message type $msgnm(rt'' X'')$ as incoming message,*
- *the variables X and Y have been declared before, with the exception that X can be the special, predefined variable **self** of type rt'.*

- *For a receive action $?msgnm(rt\,X)$, the role type rt' supports the message type $msgnm(rt\,X)$ as incoming message.*
- *State labels are unique within the process expression P.*
- *State labels are not the first action of the process expressions P_1 or P_2 in the nondeterministic choice $P_1 + P_2$.*

Building on process expressions, we can now define role behavior declarations. Opposed to full HELENA, a role behavior declaration can not invoke other processes, but can invoke itself recursively.

Definition 12 (Role Behavior Declaration). *Let Σ be an ensemble structure and rt be a role type in Σ. A role behavior declaration for rt has the form* **roleBehavior** $rt = P$ *where P is a process expression which is well-formed for rt w.r.t. Σ such that P contains (recursive) process invocations at most for rt.*[4]

Ensemble Specifications: An ensemble specification consists, as in full HELENA, of two parts: an ensemble structure and a set of role behavior declarations for all role types occurring in the ensemble structure.

Definition 13 (Ensemble specification). *An ensemble specification is a pair $EnsSpec = (\Sigma, behaviors)$ such that Σ is an ensemble structure and behaviors is a set of role behavior declarations which contains exactly one declaration* **roleBehavior** $rt = P$ *for each role type $rt \in \Sigma$.*

P2P Example: A simplified variant of the p2p example of Sec. 3, written in HELENATEXT [13], is shown in Fig. 4. In contrast to the specification in full HELENA, we omit the underlying component type `Peer` and all role attributes as well as data parameters. The ensemble structure names the participating role types and their capacity, but no multiplicities. HELENALIGHT also restricts the specification of dynamic behavior. Process expressions can only use nondeterministic choice instead of conditional selection. Thus, in contrast to the router behavior in full HELENA (cf. Fig. 3), the router *nondeterministically* either provides the file or forwards the request (cf. line 16-22 in Fig. 4b).

4.2 Semantics of HELENALIGHT Ensemble Specifications

The semantic domain of ensemble specifications are labeled transition systems describing the evolution of ensembles. Structured operational semantics (SOS) rules define the allowed transitions. We pursue an incremental approach, similar

---

[4] Note that in the above definition we use rt also as a process name for the role behavior of the role type rt.

```
1  roleType Requester {
2    rolemsg out reqAddr(Requester req);
3    rolemsg in sndAddr(Provider prov);
4    rolemsg out reqFile(Requester req);
5    rolemsg in sndFile(Provider prov);
6  }
7
8  roleType Router {
9    rolemsg in/out reqAddr(Requester req);
10   rolemsg out sndAddr(Provider prov);
11 }
12
13 roleType Provider {
14   rolemsg in reqFile(Requester req)();
15   rolemsg out sndFile(Provider prov);
16 }
17
18 ensembleStructure TransferEnsemble {
19   roleTypes = {<Requester, cap = 2>,
20                <Router,    cap = 2>,
21                <Provider,  cap = 1>};
22 }
```

```
1  roleBehavior Requester =
2    router <- create(Router) .
3    router ! reqAddr(self) .
4    ? sndAddr(Provider prov) .
5    prov ! reqFile(self) .
6    ? sndFile(Provider prov2) .
7    stateSndFile . nil
8
9  roleBehavior Provider =
10   ? reqFile(Requester req) .
11   stateReqFile .
12   req ! sndFile(self) . nil
13
14 roleBehavior Router =
15     ? reqAddr(Requester req) .
16     { prov <- create(Provider) .
17       req ! sndAddr(prov) .
18       nil }
19     +
20     { router <- create(Router) .
21       router ! reqAddr(req) .
22       Router }
```

(a) Role types and ensemble structure   (b) Role behavior declarations

Fig. 4: The p2p example in HelenaLight

to [9] and [20], by splitting the semantics into two different layers. The first
layer describes how a single role behavior evolves according to the constructs for
process expressions of the last section. The second layer builds on the first one
by defining the evolution of a whole ensemble from the concurrent evolution of
its constituent role instances.

Evolution of Roles: On the first level, we do not have any information
about the global state of the whole ensemble (involving all active role instances).
Therefore, we only formalize the progress of a single role behavior given by a
process expression. Fig. 5 defines the SOS rules inductively over the structure of
process expressions in Def. 10. Note that the rule for process invocation relies on
a given role behavior declaration. We use the symbol $\hookrightarrow$ to describe transitions
on this level. Since it does not involve instances and considers just the behavior
of single role types, this level concerns behavioral types.

Evolution of Ensembles: On the next level, we consider global states,
which we call *ensemble states*, and the concurrent execution of role instances.
For the semantics of an ensemble specification $EnsSpec = (\Sigma, behaviors)$, we
describe the possible evolutions of ensemble states (for any admissible, initial
ensemble state). An ensemble state captures the set of currently existing role
instances together with their local states. Transitions between those ensemble
states, denoted by the symbol $\rightarrow$, describe the evolution of an ensemble. They
are initiated by the actions for sending and receiving messages, role instance
creation, and state labels. According to the specified capacity of input queues
(for roles in ensemble structures), we use bounded asynchronous communication
for message exchange between role instances, i.e., each role instance has exactly
one (bounded) input queue which receives the messages issued by (other) role
instances and directed to the current one.

11

(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 invocation)	$\dfrac{Q \overset{a}{\hookrightarrow} Q'}{rt \overset{a}{\hookrightarrow} Q'}$	if **roleBehavior** $rt = Q$

Fig. 5: SOS rules for the evolution of process expressions in HELENALIGHT

Let us now look more closely to the formal definition of an ensemble state. Intuitively, an ensemble state describes the local states of all participating roles. Formally, a local state of a role instance is a tuple (rt, v, q, P) which stores the following information: the (non modifiable) role type rt of the instance, a local environment function v mapping local variables to values (the empty environment is denoted by $\emptyset$), the current content q of the input queue of the instance (the empty queue is denoted by ϵ, the length of q is denoted by $|q|$), and a process expression P representing the current control state of the instance.

We furthermore assume that each role instance has a unique identifier, represented by a positive natural number. Hence, an ensemble state representing the local states of all currently existing role instances is given by a finite function $\sigma : \mathbb{N}^+ \to \mathcal{L}$, such that $\mathcal{L}$ is the set of local states explained before. Finiteness of σ means that there exists $n \in \mathbb{N}$, denoted by $size(\sigma)$, such that $\sigma(i) = \perp$ for all $i > n$ and $\sigma(i) \neq \perp$ for all $0 < i \leq n$.[5] The definition domain of σ is denoted by $dom(\sigma)$. For instance creation, an ensemble state σ is extended by a new role instance together with its local state by assigning an element $\lambda \in \mathcal{L}$ to the next free identifier, which is $size(\sigma) + 1$ and denoted by $next(\sigma)$ in the following. Such an extension is denoted by $\sigma_{[next(\sigma) \mapsto \lambda]}$. We can also update the value of an identifier $i < next(\sigma)$ with a new value λ which is denoted by $\sigma_{[i \mapsto \lambda]}$. In summary, an ensemble state associates a local state to each currently existing role instance. Thus, a role instance i is characterized by a unique identifier and its associated local state $\lambda \in \mathcal{L}$. In the following, we often write i synonymously for the role instance identifier.

For a given ensemble specification $EnsSpec = (\Sigma, behaviors)$, the allowed transitions between ensemble states, denoted by $\to$, are described by the SOS rules in Fig. 6. For each rule, the transition between two ensemble states is inferred from a transition of process expressions on the type level, denoted by $\hookrightarrow$ in Fig. 5. The rules concern state changes of existing role instances in accordance to communication actions, the creation of new role instances (which start

---

[5] Here and in the following, we assume that the range of a finite function is implicitly extended by the undefined value $\perp$.

execution in the initial state of the behavior of their corresponding role type) and state label actions. The labels on the transitions of $\rightarrow$ indicate which role instance i currently executes which action from its role behavior specification.

$$
\text{(send)} \quad \frac{P_i \xleftarrow{Y!msgnm(X)} P_i'}{\sigma \xrightarrow{i:Y!msgnm(X)} \sigma[i \mapsto (rt_i, v_i, q_i, P_i')][j \mapsto (rt_j, v_j, q_j \cdot msgnm(k), P_j)]}
$$

$\textbf{if } i \in dom(\sigma), \sigma(i) = (rt_i, v_i, q_i, P_i),$
$\quad v_i(Y) = j \in dom(\sigma), \sigma(j) = (rt_j, v_j, q_j, P_j),$
$\quad |q_j| < roleconstraints(rt_j), v_i(X) = k \in dom(\sigma).$

$$
\text{(receive)} \quad \frac{P_i \xleftarrow{?msgnm(rt_j\ X)} P_i'}{\sigma \xrightarrow{i:?msgnm(rt_j\ X)} \sigma[i \mapsto (rt_i, v_i[X \mapsto j], q_i, P_i')]}
$$

$\textbf{if } i \in dom(\sigma), \sigma(i) = (rt_i, v_i, msgnm(j) \cdot q_i, P_i),$
$\quad j \in dom(\sigma), \sigma(j) = (rt_j, v_j, q_j, P_j).$

$$
\text{(create)} \quad \frac{P_i \xleftarrow{X \leftarrow \textbf{create}(rt_j)} P_i'}{\sigma \xrightarrow{i:X \leftarrow \textbf{create}(rt_j)} \sigma'}
$$

$\textbf{if } \sigma' = \sigma[i \mapsto (rt_i, v_i[X \mapsto next(\sigma)], q_i, P_i')][next(\sigma) \mapsto (rt_j, \emptyset[\textbf{self} \mapsto next(\sigma)], \varepsilon, P_j)],$
$\quad i \in dom(\sigma), \sigma(i) = (rt_i, v_i, q_i, P_i), \textbf{roleBehavior } rt_j = P_j.$

$$
\text{(label)} \quad \frac{P_i \xleftarrow{label} P_i'}{\sigma \xrightarrow{i:label} \sigma[i \mapsto (rt_i, v_i, q_i, P_i')]}
$$

$\textbf{if } i \in dom(\sigma), \sigma(i) = (rt_i, v_i, q_i, P_i).$

Fig. 6: SOS rules for the evolution of ensembles in HELENALIGHT

Initial States: An ensemble state σ is an *admissible initial state* for the ensemble specification *EnsSpec*, if for all $i \in dom(\sigma), \sigma(i) = (rt, \emptyset[\textbf{self} \mapsto i], \varepsilon, P)$ such that P is the process expression used in the declaration of the role behavior for rt, i.e., *EnsSpec* contains the declaration **roleBehavior** $rt = P$.

Well-Definedness of Ensemble States: A HELENALIGHT ensemble state $\sigma : \mathbb{N}^+ \rightarrow \mathcal{L}$ is well-defined if for all $i \in \mathbb{N}^+$ and $\sigma(i) = (rt, v, q, P)$:

– $\textbf{self} \in dom(v)$,
– for any (local) variable $X \in dom(v)$: $v(X) \in dom(\sigma)$,
– for $q = msgnm_1(k_1) \cdot \ldots \cdot msgnm_m(k_m)$: $k_1, \ldots, k_m \in dom(\sigma)$,

Well-definedness is not a restriction since any admissible initial state is well-defined and the SOS rules of HELENALIGHT preserve well-definedness. This follows from the syntactic restriction for well-formed role behavior declarations.

Semantics: The rules in Fig. 6 generate, for an ensemble specification *EnsSpec* and any admissible initial ensemble state σ_{init}, a labeled transition system $T_{\text{HEL}} = (S_{\text{HEL}}, I_{\text{HEL}}, A_{\text{HEL}}, \rightarrow_{\text{HEL}})$ with $I_{\text{HEL}} = \{\sigma_{init}\}$.

4.3 LTL for HelenaLight

To express goals over HelenaLight ensemble specifications, we use a subset of the LTL formulae defined for full Helena in Sec. 3. We omit atomic propositions involving attributes and only use state label propositions of the form $rt[i]@label$ where rt is a role type, $i \in \mathbb{N}^+$ and $label$ is a state label. Therefore, the set $AP(EnsSpec)$ of all atomic propositions for a HelenaLight ensemble specification $EnsSpec$ consists of all such state label expressions $rt[i]@label$. LTL formulae are built over these propositions as explained in Sec. 3.

An atomic proposition $p = rt[i]@label$ is satisfied in an ensemble state s, written $s \models p$, if there exists a role instance i of type rt whose next performed action in T_{Hel} is the state label $label$. This is well-defined since, due to well-formedness, labels are not allowed as first actions in branches. The LTS T_{Hel} for a given HelenaLight ensemble specification and an admissible initial ensemble state together with the above set $AP(EnsSpec)$ of atomic propositions and the satisfaction relation $s \models p$ induces a Kripke structure which is denoted by $K(T_{\mathrm{Hel}})$ (cf. Sec. 2). Following Def. 5, we define satisfaction of LTL in HelenaLight as follows: The LTS T_{Hel} for an ensemble specification $EnsSpec$ and an admissible initial state satisfies an LTL formulae ϕ over the set $AP(EnsSpec)$, written $T_{\mathrm{Hel}} \models \phi$, if $K(T_{\mathrm{Hel}}) \models \phi$.

P2P Example: We reformulate the goal from Sec. 3 to

$$\Box(Provider@stateReqFile \Rightarrow \Diamond Requester@stateSndFile).^{[6]}$$

In HelenaLight, we omit component types and cannot refer to attributes. Therefore, we express that the file exists in the network by the provider reaching its state labeled by *stateReqFile* (note that we have added $\Box$ since this state label expression does not hold in the initial state). Similarly, we express that the file was transferred to the requester by the requester reaching its state labeled by *stateSndFile*.

5   PromelaLight

Promela [11] is a language for modeling systems of concurrent processes. Its most important features are the dynamic creation of processes and support for synchronous and asynchronous communication via message channels. Promela verification models serve as input for the model-checker Spin [11]. On the one hand, Spin can be used to run a randomized simulation of the model. On the other hand, it can check LTL properties, formulated over a Promela specification, and find and display counterexamples.

To verify LTL properties for Helena specifications, we exploit Promela and Spin. We first translate a Helena specification to Promela and then check the specified LTL properties with Spin. Dynamic role creation in Helena can easily be expressed by dynamic process creation in Promela, and asynchronous

---

[6] *Provider@stateReqFile* and *Requester@stateSndFile* are shorthand notations without identifier which can only be used if there exists at most one instance of the role type.

message exchange between roles in HELENA by asynchronous communication via message channels in PROMELA. For formally proving the correctness of the translation, we use HELENALIGHT and for the target of the translation into PROMELA, we use an appropriate sub-language which we call PROMELALIGHT.

5.1 Syntax of PROMELALIGHT Specifications

The following syntax is a simplified version of the PROMELA syntax defined in [20]. The constructs specify a significant sub-language of the Promela definition which is sufficient as a target for the translation of HELENALIGHT.

PROMELALIGHT Specifications: Intuitively, a PROMELALIGHT specification consists of a set of process types whose behavior is specified by process expressions. We first define process expressions in PROMELALIGHT based on [20]. We use the same names for nonterminals as in [20], but sometimes we unfold the original definitions to get a smaller grammar for our purposes. In contrast to [20], we added the PROMELA expression **skip** as an explicit construct (corresponding to **nil** in HELENALIGHT). Furthermore, the conditional statement and the **goto** statement are not treated as process steps, but as a processes itself. Consequently, gotos can only occur at the end of a process expression. We have also removed guards from the conditional statements, thus obtaining nondeterministic choice.

Definition 14 (Process Expressions). *A process expression seq is built from the following grammar, where label is the name of a state label (used for gotos and verification), var, var_1, and var_2 are names of variables, const is a constant, pt is the name of a process type, and typelist is a comma-separated list of types:*

$seq ::=$	$\textbf{skip}$	*(empty process)*
	$\mid step; seq$	*(sequential composition)*
	$\mid \textbf{if} :: seq_1 :: seq_2\, \textbf{fi}$	*(nondeterministic choice)*
	$\mid \textbf{goto}\ label$	*(goto)*
$step ::=$	$label : \textbf{true}$	*(state label)*
	$\mid var_1!const, var_2$	*(send)*
	$\mid var_1?const, var_2$	*(receive)*
	$\mid \textbf{run}\ pt(var)$	*(run)*
	$\mid \textbf{chan}\ var$	*(channel declaration)*
	$\mid \textbf{chan}\ var = [const]\ \textbf{of}\ \{typelist\}$	*(channel declaration with initialization)*

Note that send and receive steps always concern data tuples $const, var_2$ consisting of a constant and a variable. A channel declaration **chan** $var = \dots$ opens the scope for a local channel variable var. We assume that the names of the declared variables are unique within a process expression and different from **self**, which is a predefined variable of type **chan** that can always be used. A variable is *initialized* if either the variable occurs in a receive step as var_2 or in a channel declaration with initialization as var or is the special variable **self**.

15

Definition 15 (Well-Formedness of Process Expressions). *A process expression is* well-formed *if (1) all variables occurring in a send or run step have been initialized before, (2) the variable var_1 in a receive step has been initialized before and the variable var_2 has been declared before, and (3) label :* **true** *is not the first statement in seq_1 or seq_2 in* **if** *:: seq_1 :: seq_2* **fi***.*

Process expressions are used to define process types. In PROMELALIGHT, a process type has always one parameter **self** of type **chan** which represents a distinguished input channel for each process instance.

Definition 16 (Process Type Declaration). *A process type declaration has the form* **proctype** *pt(***chan self***)$\{start_{pt}$:* **true**$; seq\}$ *where pt is the name of the process type, seq is a well-formed process expression not containing a state label $start_{pt}$:* **true***, and any* **goto** *expression occurring in seq has the form* **goto** *$start_{pt}$.*

The above definition associates a process expression to a process type *pt*. It allows a restricted version of recursion by introducing the state label $start_{pt}$: **true** at the beginning of the process and allowing to jump back to that via **goto** $start_{pt}$. This syntactic restriction simplifies the semantics since the continuation of a **goto** is then uniquely determined. Hence, we do not need to carry the full body of a process type declaration in the semantic states and to search for labels in the body to find the continuation as in [20].

Definition 17 (PROMELALIGHT specification). *A* PROMELALIGHT *specification consists of a set of process type declarations.*

P2P Example: The formal translation from HELENALIGHT to PROMELALIGHT will be discussed in Sec. 7. To illustrate PROMELALIGHT, we already present here, in Fig. 7, the PROMELALIGHT translation of the simplified variant of the p2p example. Let us briefly look at the process type declaration for a router in Fig. 7b in comparison to the role behavior declaration in Fig. 4b. Non-deterministic choice is expressed by the **if** construct of PROMELALIGHT. Role instance creation in HELENALIGHT is translated to starting a new process in PROMELALIGHT (line 10 and 15 in Fig. 4b). Asynchronous message exchange is obtained by passing an asynchronous channel to the newly created process for communication (line 9 and 14 in Fig. 4b).

5.2 Semantics of PROMELALIGHT Specifications

The semantic domain of PROMELALIGHT specifications are again labeled transition systems. We also follow a two-level SOS approach which has been advocated for the formal PROMELA semantics in [20]. On the first level, the SOS rules only deal with the progress of process expressions specified by the nonterminal symbol *seq* in Def. 14. Process instances and their concurrent execution are considered on the second level.

Evolution of Process Expressions: On the first level, we only formalize the progress determined by a single process expression. Fig. 8 defines the SOS

```
1  mtype { reqAddr, sndAddr,              1  proctype Router(chan self) {
2         reqFile, sndFile }              2    startRouter: true;
3  proctype Requester(chan self) {        3
4    startRequester: true;                4    chan req;
5    chan router = [2] of { mtype, chan };5    self?reqAddr,req;
6    run Router(router);                  6
7    router!reqAddr,self;                 7    if
8    chan prov;                           8    ::
9    self?sndAddr,prov;                   9      chan prov = [1] of { mtype, chan };
10   prov!reqFile,self;                   10     run Provider(prov);
11   chan prov2;                          11     req!sndAddr,prov;
12   self?sndFile,prov2;                  12     skip
13   stateSndFile: true;                  13    ::
14   skip                                 14     chan router = [2] of { mtype, chan };
15 }                                      15     run Router(router);
16 proctype Provider(chan self) {         16     router!reqAddr,req;
17   startProvider: true;                 17     goto startRouter
18   chan req;                            18    fi
19   self?reqFile,req;                    19 }
20   stateReqFile: true;                  20 init {
21   req!sndFile,self;                    21   chan req = [2] of { mtype, chan };
22   skip                                 22   run Requester(req);
23 }                                      23 }
```

(a) Message definitions and process type   (b) Process type declaration for `Router`
declarations for `Requester` and `Provider`

Fig. 7: The p2p example in PROMELALIGHT

rules inductively over the structure of PROMELALIGHT process expressions in Def. 14 where the symbol $\hookrightarrow$ describes transitions on this level. In contrast to [20], we postpone not only the treatment of process instances, but also the treatment of local environments and the consideration of channel instances to the second level.

$$(\text{sequential composition}) \quad step; seq \xrightarrow{step} seq$$

$$(\text{choice-left}) \quad \frac{seq_1 \xrightarrow{step} seq_1'}{\textbf{if} :: seq_1 :: seq_2 \, \textbf{fi} \xrightarrow{step} seq_1'}$$

$$(\text{choice-right}) \quad \frac{seq_2 \xrightarrow{step} seq_2'}{\textbf{if} :: seq_1 :: seq_2 \, \textbf{fi} \xrightarrow{step} seq_2'}$$

$$(\text{goto}) \quad \textbf{goto} \, \text{start}_{pt} \xrightarrow{\textbf{goto start}_{pt}} \text{start}_{pt} : \textbf{true}; seq$$
$$\textbf{if proctype} \, pt(\textbf{chan self})\{\text{start}_{pt} : \textbf{true}; seq\}$$

Fig. 8: SOS rules for the evolution of a process expression in PROMELALIGHT

Evolution of Concurrent Process Instances: On the next level, we consider global states and the concurrent execution of process instances. Similarly to ensemble states in HELENALIGHT, a global state in PROMELALIGHT captures the currently existing process instances. However, in contrast to input queues

17

in HELENALIGHT, process instances communicate via channels which are not owned by a local process, but belong to the global state. Hence, a global state of a PROMELALIGHT specification captures (1) the set of the currently existing channel instances (together with their states) and (2) the set of the currently existing process instances (together with their local states). Transitions between global states are initiated by the actions for sending and receiving a message, running a new process, channel declarations, **goto**s, and state labels.

Let us now look more closely to the formal definition of a global state in PROMELALIGHT. Intuitively, a global state describes the local states of all currently existing channels and the local states of all currently existing process instances. Each channel instance is uniquely identified by a positive natural number and the currently existing channel instances are represented by a finite function (called *channel function*) $\mathsf{ch} : \mathbb{N}^+ \to \mathcal{C}$ such that $\mathcal{C}$ is the set of local channel states. A local state of a channel is a tuple (T, ω, κ) consisting of the (non-modifiable) type T of entries, the content ω which is a word of T-values (we write ε for the empty word), and the (non-modifiable) capacity $\kappa > 0$ of the channel[7]. Similarly, each process instance is uniquely identified by a positive natural number[8], and the currently existing process instances are represented by a finite function $\mathsf{proc} : \mathbb{N}^+ \to \mathcal{P}$ such that $\mathcal{P}$ is the set of local process (instance) states. A local state of a process instance is a tuple (pt, β, π) where pt is the process type of the instance, β is a local environment function mapping local variables to values (i.e., channel identifiers or **null**) and π is a process expression representing the current control state of the instance. Finally, a global state is a pair $(\mathsf{ch}, \mathsf{proc})$ of a channel function ch and a function proc representing the currently existing process instances.[9]

For a given PROMELALIGHT specification, the allowed transitions between global states, denoted by $\to$, are described by the SOS rules in Fig. 9. They evolve a set of process instances which execute in accordance with their process types under the assumption of asynchronous communication. For each rule, the transition between two global states is inferred from a transition of process expressions on the type level, denoted by $\hookrightarrow$ in Fig. 8. The labels on the transitions of $\to$ indicate which process instance i currently executes which step from its process type specification. In the rules, we use the shorthand notations for the extension and update of finite functions from Sec. 4.2.

Initial States: A global state $(\mathsf{ch}, \mathsf{proc})$ is an *admissible initial state* for a PROMELALIGHT specification, if

– for all $c \in dom(\mathsf{ch})$: $\mathsf{ch}(c) = (T, \varepsilon, \kappa)$ for some T and κ,
– for all $i \in dom(\mathsf{proc})$: $\mathsf{proc}(i) = (pt, \emptyset[\mathbf{self} \mapsto c_i], \mathrm{start}_{pt} : \mathbf{true}; seq)$ such that $c_i \in dom(\mathsf{ch})$ with $c_i \neq c_j$ for $i \neq j$ and the PROMELALIGHT specification contains the process type declaration **proctype** $pt(\mathbf{chan\ self})\{\mathrm{start}_{pt} : \mathbf{true}; seq\}$.

---

[7] In PROMELALIGHT, we only consider asynchronous communication ($\kappa > 0$).

[8] For technical reasons, explained in the discussion of initial states below, we deviate from [20] and do not use 0 as an identifier for channels and processes.

[9] In [20], ch is denoted by $\mathcal{C}$, proc by act, and β by $\mathcal{L}$.

$$(\text{goto}) \qquad \cfrac{\pi_i \xrightarrow{\ \mathbf{goto}\ label\ } \pi_i'}{(\mathsf{ch}, \mathsf{proc}) \xrightarrow{\ i:\mathbf{goto}\ label\ } (\mathsf{ch}, \mathsf{proc}_{[i \mapsto (pt_i, \beta_i, \pi_i')]})}$$

if $i \in dom(\mathsf{proc}), \mathsf{proc}(i) = (pt_i, \beta_i, \pi_i)$.

$$(\text{label}) \qquad \cfrac{\pi_i \xrightarrow{\ label:\mathbf{true}\ } \pi_i'}{(\mathsf{ch}, \mathsf{proc}) \xrightarrow{\ i:label:\mathbf{true}\ } (\mathsf{ch}, \mathsf{proc}_{[i \mapsto (pt_i, \beta_i, \pi_i')]})}$$

if $i \in dom(\mathsf{proc}), \mathsf{proc}(i) = (pt_i, \beta_i, \pi_i)$.

$$(\text{send}) \qquad \cfrac{\pi_i \xrightarrow{\ var_1!const, var_2\ } \pi_i'}{(\mathsf{ch}, \mathsf{proc}) \xrightarrow{\ i:var_1!const, var_2\ } (\mathsf{ch}_{[c \mapsto (T, \omega \cdot (const, v), \kappa)]}, \mathsf{proc}_{[i \mapsto (pt_i, \beta_i, \pi_i')]})}$$

if $i \in dom(\mathsf{proc}), \mathsf{proc}(i) = (pt_i, \beta_i, \pi_i), \beta_i(var_2) = v \in dom(\mathsf{ch})$,
$\quad \beta_i(var_1) = c \in dom(\mathsf{ch}), \mathsf{ch}(c) = (T, \omega, \kappa), |\omega| < \kappa$.

$$(\text{receive}) \qquad \cfrac{\pi_i \xrightarrow{\ var_1?const, var_2\ } \pi_i'}{(\mathsf{ch}, \mathsf{proc}) \xrightarrow{\ i:var_1?const, var_2\ } (\mathsf{ch}_{[c \mapsto (T, \omega, \kappa)]}, \mathsf{proc}_{[i \mapsto (pt_i, \beta_i[var_2 \mapsto v], \pi_i')]})}$$

if $i \in dom(\mathsf{proc}), \mathsf{proc}(i) = (pt_i, \beta_i, \pi_i), var_2 \in dom(\beta_i)$,
$\quad \beta_i(var_1) = c \in dom(\mathsf{ch}), \mathsf{ch}(c) = (T, (const, v) \cdot \omega, \kappa), v \in dom(\mathsf{ch})$.

$$(\text{run}) \qquad \cfrac{\pi_i \xrightarrow{\ \mathbf{run}\ pt_j(var)\ } \pi_i'}{(\mathsf{ch}, \mathsf{proc}) \xrightarrow{\ i:\mathbf{run}\ pt_j(var)\ } \gamma'}$$

if $\gamma' = (\mathsf{ch}, \mathsf{proc}_{[i \mapsto (pt_i, \beta_i, \pi_i')]}[next(\mathsf{proc}) \mapsto (pt_j, \emptyset[\mathbf{self} \mapsto c], \mathrm{start}_{pt_j}:\mathbf{true}; seq)])$
$\quad i \in dom(\mathsf{proc}), \mathsf{proc}(i) = (pt_i, \beta_i, \pi_i), \beta_i(var) = c \in dom(\mathsf{ch})$,
$\quad \mathbf{proctype}\ pt_j(\mathbf{chan\ self})\{\mathrm{start}_{pt_j} : \mathbf{true}; seq\}$.

$$(\text{chan-1}) \qquad \cfrac{\pi_i \xrightarrow{\ \mathbf{chan}\ var\ } \pi_i'}{(\mathsf{ch}, \mathsf{proc}) \xrightarrow{\ i:\mathbf{chan}\ var\ } (\mathsf{ch}, \mathsf{proc}_{[i \mapsto (pt_i, \beta_i[var \mapsto \mathbf{null}], \pi_i')]})}$$

if $i \in dom(\mathsf{proc}), \mathsf{proc}(i) = (pt_i, \beta_i, \pi_i)$

$$(\text{chan-2}) \qquad \cfrac{\pi_i \xrightarrow{\ \mathbf{chan}\ var=[const]\ \mathbf{of}\ \{typelist\}\ } \pi_i'}{(\mathsf{ch}, \mathsf{proc}) \xrightarrow{\ i:\mathbf{chan}\ var=\ldots\ } \gamma'}$$

if $\gamma' = (\mathsf{ch}_{[next(\mathsf{ch}) \mapsto (typelist, \varepsilon, const)]}, \mathsf{proc}_{[i \mapsto (pt_i, \beta_i[var \mapsto next(\mathsf{ch})], \pi_i')]})$
$\quad i \in dom(\mathsf{proc}), \mathsf{proc}(i) = (pt_i, \beta_i, \pi_i)$.

Fig. 9: SOS rules for the evolution of concurrent process instances.

Concrete initial states in PROMELALIGHT are constructed by running an appropriate initialization as shown in line 20-23 of Fig. 7b where one channel and one requester instance, using that channel as input, are created. The initialization is executed by a root process `init` which implicitly obtains the identifier 0. However, we do not consider this process in a PROMELALIGHT specification and are not interested in the verification of properties for the root process (which anyway does not have any counterpart in a HELENA specification). Thus, we

19

use in our semantic framework and in atomic propositions of LTL formulae only positive natural numbers for process identifiers.

Well-Definedness of Global States: A global PROMELALIGHT state $\gamma = (\mathsf{ch}, \mathsf{proc})$ with $\mathsf{ch} : \mathbb{N}^+ \to \mathcal{C}$ and $\mathsf{proc} : \mathbb{N}^+ \to \mathcal{P}$ is well-defined if for all $i \in \mathbb{N}^+$, $\mathsf{ch}(i) = (T, \omega, \kappa)$ and $\mathsf{proc}(i) = (pt, \beta, \pi)$:

- $\beta(\mathbf{self}) \in dom(\mathsf{ch})$,
- for any (local) variable $X \in dom(\beta)$: $\beta(X) \in dom(\mathsf{ch}) \cup \{\mathbf{null}\}$,
- for $\omega = (msgnm_1, c_1) \cdot \ldots \cdot (msgnm_m, c_m)$: $c_1, \ldots, c_m \in dom(\mathsf{ch})$ and $\kappa \geq m$.

Semantics: As for HELENALIGHT, the rules in Fig. 9 generate, for a PROMELALIGHT specification and any admissible initial state γ_{init}, a labeled transition system $T_{\text{PRM}} = (S_{\text{PRM}}, I_{\text{PRM}}, A_{\text{PRM}}, \to_{\text{PRM}})$ with $I_{\text{PRM}} = \{\gamma_{init}\}$.

5.3 LTL for PROMELALIGHT

To express goals over PROMELALIGHT specifications, we use LTL formulae. As in HELENALIGHT, we restrict the atomic propositions of LTL formulae to state label expressions of the form $pt[i]@label$ where pt is a process type, $i \in \mathbb{N}^+$ and $label$ is a state label. Therefore, the set $AP(PrmSpec)$ of all atomic propositions for a PROMELALIGHT specification $PrmSpec$ consists of all such state label expressions $pt[i]@label$. LTL formulae are built over these propositions as explained in Sec. 3.

An atomic proposition $p = pt[i]@label$ is satisfied in a global state γ, written $\gamma \models p$ if there exists a process instance i of type pt whose next performed action in T_{PRM} is the state label $label$. The LTS T_{PRM} for a given PROMELALIGHT specification and an admissible initial state together with the above set $AP(PrmSpec)$ of atomic propositions and the satisfaction relation $\gamma \models p$ induces a Kripke structure denoted by $K(T_{\text{PRM}})$ (cf. Sec. 2). Following Def. 5, we define satisfaction of LTL in PROMELALIGHT as follows: The LTS T_{PRM} for a PROMELALIGHT specification $PrmSpec$ and an admissible initial state satisfies an LTL formulae ϕ over the set $AP(PrmSpec)$, written $T_{\text{PRM}} \models \phi$, if $K(T_{\text{PRM}}) \models \phi$.

6 Translation of HELENALIGHT to PROMELALIGHT

In this section, we propose a transformation from HELENALIGHT ensemble specifications to PROMELALIGHT verification models. We assume given a HELENALIGHT ensemble specification $EnsSpec = (\Sigma, behaviors)$ with $\Sigma = (nm, roletypes, roleconstraints)$ being an ensemble structure. The translation into the PROMELALIGHT specification $trans(EnsSpec)$ proceeds in two steps: First, we provide all message types from HELENALIGHT in PROMELALIGHT by declaring an enumeration type, called **mtype** (not shown here).

Translation of Role Behavior Declarations: Then, for each role type and its corresponding role behavior in HELENALIGHT, we create a process type in PROMELALIGHT which reflects the execution of the role behavior and is inductively defined over the structure of process expressions and actions.

$$
\begin{aligned}
trans_{\mathrm{decl}}(\textbf{roleBehavior}\,rt{=}P) \quad &= \textbf{proctype}\,rt(\textbf{chan self})\;\{ \\
&\qquad \mathrm{start}_{rt} : \textbf{true};\, trans_{\mathrm{proc}}(P)\;\}
\end{aligned}
$$

$$
\begin{aligned}
trans_{\mathrm{proc}}(\textbf{nil}) \quad &= \textbf{skip} \\
trans_{\mathrm{proc}}(a.P) \quad &= trans_{\mathrm{act}}(a);\, trans_{\mathrm{proc}}(P) \\
trans_{\mathrm{proc}}(P_1 + P_2) \quad &= \textbf{if} :: trans_{\mathrm{proc}}(P_1) :: trans_{\mathrm{proc}}(P_2)\,\textbf{fi} \\
trans_{\mathrm{proc}}(N) \quad &= \textbf{goto}\,\mathrm{start}_N
\end{aligned}
$$

$$
\begin{aligned}
trans_{\mathrm{act}}(Y!msgnm(X)) \quad &= Y!msgnm, X \\
trans_{\mathrm{act}}(?msgnm(rt\,X)) \quad &= \textbf{chan}\,X;\, \textbf{self}?msgnm, X \\
trans_{\mathrm{act}}(X \leftarrow \textbf{create}(rt)) \quad &= \textbf{chan}\,X = [_{roleconstraints(rt)}]\,\textbf{of}\,\{\textbf{mtype},\textbf{chan}\}; \\
&\qquad \textbf{run}\,rt(X) \\
trans_{\mathrm{act}}(label) \quad &= label : \textbf{true} \\
\textbf{if}\,label \neq \mathrm{start}_{rt}
\end{aligned}
$$

As an example, we consider the HELENALIGHT specification in Fig. 4 which is translated with the above rules to the PROMELALIGHT specification in Fig. 7.

Translation of Initial States: To be able to show semantic equivalence between HELENALIGHT and PROMELALIGHT specifications, we have to translate admissible initial states. We assume given an admissible initial HELENALIGHT ensemble state σ with local states $\sigma(i) = (rt, \emptyset[\textbf{self} \mapsto i], \varepsilon, P)$ for all $i \in dom(\sigma)$. Its translation is the admissible initial PROMELALIGHT state $trans_{\mathrm{init}}(\sigma) = (\textsf{ch}, \textsf{proc})$, such that the content of all existing channels in $\textsf{ch}$ is empty, $dom(\textsf{proc}) = dom(\sigma)$, and for all $i \in dom(\textsf{proc})$, $\textsf{proc}(i) = (rt, \emptyset[\textbf{self} \mapsto c_i], \mathrm{start}_{rt} : \textbf{true}; trans_{\mathrm{proc}}(P))$ with $c_i \in dom(\textsf{ch})$ and $c_i \neq c_j$ for $i \neq j$.

7 ≈-Stutter Equivalence of the Translation

We now prove the correctness of the translation from HELENALIGHT to PROMELALIGHT, i.e., that a HELENALIGHT specification and its PROMELALIGHT translation satisfy the same set of $LTL_{\backslash \mathbf{X}}$ formulae. We first define two relations $\sim$ and $\approx$ between the Kripke structures induced from a HELENALIGHT specification and its PROMELALIGHT translation. To be able to apply Thm. 1 from Sec. 2, we show that $\approx$ preserves satisfaction of atomic propositions and any admissible initial state of a HELENALIGHT ensemble specification and its PROMELALIGHT translation are related by $\sim$. Furthermore, we prove that the relation $\sim$ is a $\approx$-stutter simulation of the Kripke structure of the HELENALIGHT specification by the Kripke structure of the PROMELALIGHT translation and the inverse relation $\approx^{-1}$ is a $\approx^{-1}$-stutter simulation in the other direction. Having proven stutter trace equivalence, we can then apply Thm. 2 entailing preservation of $LTL_{\backslash \mathbf{X}}$.

Silent Actions: To prove stutter trace equivalence of Kripke structures, we rely on the transitions in the labeled transition systems which induce the Kripke structures. Thereby, some transitions in HELENALIGHT are reflected by several transitions in PROMELALIGHT, e.g., the transition with the action $?msgnm(X)$ is reflected by two transitions with the actions $\textbf{chan}\,X$ and $\textbf{self}?msgnm, X$ (cf. definition of $trans_{\mathrm{act}}$ in Sec. 6). These additional transitions do not change

satisfaction of atomic propositions. Thus, we consider the following steps and their corresponding actions in PROMELALIGHT as silent and denote them by τ:

- the transition from **chan** X; **self**?$msgnm, X$ to **self**?$msgnm, X$,
- the transition from **chan** $X = [roleconstraints(rt)]$ **of** $\{$**mtype,chan**$\}$; **run** $rt(X)$ to **run** $rt(X)$,
- the transition $start_{pt_i}$: **true**; π to π since start state labels $start_{pt_i}$: **true** only exist in PROMELALIGHT, and
- the transition **goto** $start_{pt_i}$; π to π since in PROMELALIGHT, recursive process invocation is expressed by a jump (i.e., a **goto** step) while in HELENALIGHT, the body of the invoked role behavior is directly applied without any execution step for recursion.

Simulation Relations: We define two relations which both express a correspondence between HELENALIGHT ensemble states and global PROMELALIGHT states, but require a different level of correspondence.

Definition 18 (Relation $\sim$ and $\approx$). *Let $K(T_{\text{HEL}}) = (S_{\text{HEL}}, I_{\text{HEL}}, \rightarrow^{\bullet}_{\text{HEL}}, F_{\text{HEL}})$ be the induced Kripke structure of a HELENALIGHT ensemble specification and $K(T_{\text{PRM}}) = (S_{\text{PRM}}, I_{\text{PRM}}, \rightarrow^{\bullet}_{\text{PRM}}, F_{\text{PRM}})$ be the induced Kripke structure of a PROMELALIGHT specification. The relation $\sim \, \subseteq S_{\text{HEL}} \times S_{\text{PRM}}$ is defined as follows: $\sigma \sim (\boldsymbol{ch}, \boldsymbol{proc})$ if it holds that*

1. *$dom(\sigma) = dom(\boldsymbol{proc})$ and*
2. *for all $i \in dom(\sigma)$ with $\sigma(i) = (rt_i, v_i, q_i, P_i)$ and $\boldsymbol{proc}(i) = (pt_i, \beta_i, \pi_i)$:*
   (a) *$rt_i = pt_i$,*
   (b) *$dom(v_i) \subseteq dom(\beta_i)$ such that for all $X \in dom(v_i)$:*
       $v_i(X) = j \Leftrightarrow \beta_i(X) = \beta_j(\boldsymbol{self})$ (where $\boldsymbol{proc}(j) = (pt_j, \beta_j, \pi_j)$),
   (c) *$q_i = msgnm_1(k_1) \cdot \ldots \cdot msgnm_m(k_m) \Leftrightarrow$*
       $\boldsymbol{ch}(\beta_i(\boldsymbol{self})) = (T, (msgnm_1, \beta_{k_1}(\boldsymbol{self})) \cdot \ldots \cdot (msgnm_m, \beta_{k_m}(\boldsymbol{self})), \kappa)$,
   (d) *$trans_{proc}(P_i) = \pi_i$ or $\pi_i \xrightarrow{start_{pt_i}:\boldsymbol{true}}_{\text{PRM}} trans_{proc}(P_i)$.*

The relation $\approx \, \subseteq S_{\text{HEL}} \times S_{\text{PRM}}$ is defined just as the relation $\sim$ with the exception of item (2d) where $trans_{proc}(P_i) = \pi_i$ is replaced by $trans_{proc}(P_i) \xrightarrow{\tau^*}_{\text{PRM}} \pi_i$. Obviously, it holds that $\sim \, \subseteq \, \approx$.

Firstly, in the defined relations, there must be as many role instances in HELENALIGHT as process instances in PROMELALIGHT. Secondly, the local state of each role instance i must be related to the local state of the process instance with the same identifier i: (a) The role type rt_i must match the process type pt_i. (b) The local variables in v_i must have counterparts in β_i, but note that the value types of HELENALIGHT and PROMELALIGHT are subtly different. A local variable in HELENALIGHT points to a role instance whereas a local variable in PROMELALIGHT points to a channel. Furthermore, note that vice versa, there might be local variables in β_i which do not have any counterparts in v_i. (c) The content of the input queue of the role instance must match the content of the corresponding channel of the process instance. As for local variables, the input

queue of the role instance consists of role instance identifiers whereas the related PROMELALIGHT input channel contains the identifiers of the input channels of the process instances (corresponding to these role instances). (d) For the process expression π_i occurring in the local state of the process instance, we either require that it is the same as the translation of the process expression P_i occurring in the local state of the role instance or that it can evolve by the single action start_{pt_i} : **true** to the translation of P_i. The latter takes into account that the translation of a role behavior into PROMELALIGHT adds a start label at the beginning of the translated role behavior. For the relation $\approx$, we weaken the first condition such that π_i must only be reachable by evolving the translation of P_i with arbitrary many τ actions.

Properties of the Simulation Relations: Based on the induced Kripke structures, we show some interesting properties of the two relations: the relation $\approx$ preserves satisfaction of atomic propositions and any admissible initial state in HELENALIGHT and its PROMELALIGHT translation are related by the relation $\sim$.

Lemma 1 (Preservation of Atomic Propositions). *Let* $K(T_{\mathrm{HEL}}) = (S_{\mathrm{HEL}}, I_{\mathrm{HEL}}, \rightarrow^{\bullet}_{\mathrm{HEL}}, F_{\mathrm{HEL}})$ *be an induced Kripke structure of a* HELENALIGHT *ensemble specification* $EnsSpec = (\Sigma, behaviors)$ *such that no role behavior in behaviors starts with a state label and let* $K(T_{\mathrm{PRM}}) = (S_{\mathrm{PRM}}, I_{\mathrm{PRM}}, \rightarrow^{\bullet}_{\mathrm{PRM}}, F_{\mathrm{PRM}})$ *be the induced Kripke structure of a* PROMELALIGHT *specification.*

For all $\sigma \in S_{\mathrm{HEL}}, \gamma \in S_{\mathrm{PRM}}$, *if* $\sigma \approx \gamma$, *then* $F_{\mathrm{HEL}}(\sigma) = F_{\mathrm{PRM}}(\gamma)$.

Proof. A HELENALIGHT state σ satisfies an atomic proposition $p = rt[i]@label$ only if there exists a role instance i of type rt whose next performed action is the state label *label*; similarly, γ satisfies p only if there exists a process instance i of type rt whose next performed action is the state label *label*. In any other case p is not satisfied. For each such proposition p, the proof proceeds by induction on the depth of the derivation of the next action of the process expression for role instance i. The first interesting case in the induction is nondeterministic choice. In both, HELENALIGHT and PROMELALIGHT, labels are not allowed as first actions in a branch such that the nondeterministic choice and each branch separately do not satisfy any atomic proposition p. Therefore, the induction step is trivial. Another interesting case is process invocation. HELENALIGHT executes directly the first action of its role behavior. On the other hand, in PROMELALIGHT process invocation is realized by a **goto** jump followed by the start label and the first action of the (translated) role behavior. These steps (trivially) preserve satisfaction of atomic propositions since start labels are not allowed as atomic propositions and since the first action of a role behavior cannot be a state label. □

Lemma 2 (Relationship between Initial States). *Let* σ *be an admissible initial state of a* HELENALIGHT *ensemble specification, then* $\sigma \sim trans_{init}(\sigma)$.

Proof. In HELENALIGHT, an admissible initial state σ consists of local states $\sigma(i) = (rt, \emptyset[\mathbf{self} \mapsto i], \varepsilon, P)$ (cf. definition of admissible initial states in Sec. 4.2).

23

For the PROMELALIGHT translation holds $trans_{\text{init}}(\sigma) = (\text{ch}, \text{proc})$; the content of all existing channels in ch is empty, $dom(\text{proc}) = dom(\sigma)$, and for all $i \in dom(\text{proc})$, we have $\text{proc}(i) = (rt, \emptyset[\text{self} \mapsto c_i], \text{start}_{rt} : \textbf{true}; trans_{\text{proc}}(P))$ with $c_i \in dom(\text{ch})$ (cf. definition of $trans_{\text{init}}$ in Sec. 6). Therefore, all conditions for $\sigma \sim trans_{\text{init}}(\sigma)$ are satisfied, in particular item (2d) is satisfied since $\text{start}_{rt} : \textbf{true}; trans_{\text{proc}}(P) \xrightarrow{\text{start}_{rt}:\textbf{true}}_{\text{PRM}} trans_{\text{proc}}(P_i)$. $\qquad\square$

Stutter Simulations: Based on the previous two lemmata, we move on to show that the relation $\sim$ is a $\approx$-stutter simulation of a HELENALIGHT specification by its PROMELALIGHT translation and that the inverse relation $\approx^{-1}$ itself is a $\approx^{-1}$-stutter simulation in the other direction. Note that $\approx$ itself would not preserve the branching structure of $K(T_{\text{HEL}})$ (due to branching with silent actions in PROMELALIGHT), but the coarser relation $\sim$ does.

Proposition 1 (Stutter Simulation of HELENALIGHT Specifications).
Let $K(T_{\text{HEL}})$ and $K(T_{\text{PRM}})$ be the induced Kripke structures of a HELENALIGHT ensemble specification and of its PROMELALIGHT translation $trans(EnsSpec)$ as in Lemma 1. Then, $\sim$ is a $\approx$-stutter simulation of K_{HEL} by K_{PRM}.

Proof. With Lemma 2, we proved that any initial state of a HELENALIGHT specification and its PROMELALIGHT translation are in the relation $\sim$. It remains to show that the relation $\sim$ fulfills the property of a $\approx$-stutter simulation described in Def. 6. In the proof, we rely on the underlying labeled transition systems T_{HEL} and T_{PRM} of the Kripke structures $K(T_{\text{HEL}})$ and $K(T_{\text{PRM}})$. To reflect labels of HELENALIGHT in PROMELALIGHT, we introduce a notation which translates a HELENALIGHT label to its corresponding PROMELALIGHT label by omitting silent actions:

$$
\begin{aligned}
trans_{\text{label}}(i : Y!msgnm(X)) &= i : Y!msgnm, X \\
trans_{\text{label}}(i :?msgnm(rt_j\, X)) &= i : \textbf{self}?msgnm, X \\
trans_{\text{label}}(i : X \leftarrow \textbf{create}(rt_j)) &= i : \textbf{run}\ rt_j(X) \\
trans_{\text{label}}(i : label) &= i : label : \textbf{true}
\end{aligned}
$$

By relying on that notation, we show the following property which entails the required property for a $\approx$-stutter simulation:

For all $\sigma \in S_{\text{HEL}}, \gamma \in S_{\text{PRM}}$ with $\sigma \sim \gamma$, if $\sigma \xrightarrow{a}_{\text{HEL}} \sigma'$,
then there exists $\gamma \xrightarrow{\tau}_{\text{PRM}} \gamma_1 \ldots \xrightarrow{\tau}_{\text{PRM}} \gamma_n \xrightarrow{trans_{\text{label}}(a)}_{\text{PRM}} \gamma'\ (n \geq 0)$
such that $\sigma \approx \gamma_k$ for all $k \in \{1, \ldots, n\}$, and $\sigma' \sim \gamma'$.

The proof proceeds by induction on the depth of the derivation of $\sigma \xrightarrow{a}_{\text{HEL}} \sigma'$. The induction relies on the following fact: Each action a in HELENALIGHT is reflected in PROMELALIGHT, but some internal steps might be necessary in PROMELALIGHT before the corresponding action $trans_{\text{label}}(a)$ can actually be executed, e.g., message reception with the action $i : ?msgnm(rt_j\, X)$ is translated to two actions $i : \textbf{chan}\ X$ and $i : \textbf{self}?msgnm, X$ or process invocation in PROMELALIGHT uses first a **goto** step and then a start label step to reach the

beginning of the (translated) role behavior. Since the relation $\approx$ just requires that the translation of the HELENALIGHT process expression P for role instance i can evolve by τ actions to the PROMELALIGHT process expression π for the corresponding process instance i, all those intermediate steps result in states remaining in the relation $\approx$. Only the translated action $trans_{\text{label}}(a)$ evolves the PROMELALIGHT translation according to the evolution of the HELENALIGHT specification such that the resulting states are again in the relation $\sim$. $\square$

In the other direction, the inverse relation $\approx^{-1}$ serves as $\approx^{-1}$-stutter simulation.

Proposition 2 (Stutter Simulation of PROMELALIGHT Translations).
Let $K(T_{\text{HEL}})$ and $K(T_{\text{PRM}})$ be the induced Kripke structures of a HELENALIGHT ensemble specification and of its PROMELALIGHT translation trans(EnsSpec) as in Lemma 1. Then, $\approx^{-1}$ is a $\approx^{-1}$-stutter simulation of K_{PRM} by K_{HEL}.

Proof. We rely on Lemma 2 as before. The proof that the relation $\approx^{-1}$ satisfies the property for a $\approx^{-1}$-stutter simulation is based, as before, on the underlying labeled transition systems. We show the following property which entails the required property for a $\approx^{-1}$-stutter simulation:

$$\text{For all } \gamma \in S_{\text{PRM}}, \sigma \in S_{\text{HEL}} \text{ with } \gamma \approx^{-1} \sigma, \text{ if } \gamma \xrightarrow{b}_{\text{PRM}} \gamma',$$
$$\text{then } \gamma' \approx^{-1} \sigma \text{ if } b = \tau \text{ or there exists } \sigma \xrightarrow{a}_{\text{HEL}} \sigma' \text{ if } b \neq \tau$$
$$\text{such that } trans_{\text{label}}(a) = b \text{ and } \gamma' \approx^{-1} \sigma'.$$

The proof proceeds by induction on the depth of the derivation of $\gamma \xrightarrow{b}_{\text{PRM}} \gamma'$. The induction relies on the following fact: Silent actions in PROMELALIGHT, denoted by τ might only change the value of local variables which are not yet in relation to HELENALIGHT, i.e., silent steps preserve the relationship according to $\approx^{-1}$. For all non-silent actions, the relation $\approx^{-1}$ is sufficient to transfer executability of a PROMELALIGHT action b to its corresponding HELENALIGHT action a with $trans_{\text{label}}(a) = b$ such that $\approx^{-1}$ is again established by the transition. $\square$

Lemma 1, Prop. 1, and Prop. 2 allow us to infer, by Thm. 1, that the induced Kripke structures of a HELENALIGHT ensemble specification and its PROMELA-LIGHT translation are stutter trace equivalent. Thus, we can apply Thm. 2 to show that the both labeled transitions systems satisfy the same $LTL_{\backslash \mathbf{X}}$ formulae.

Theorem 3 (HELENALIGHT $LTL_{\backslash \mathbf{X}}$ Preservation). *Let T_{HEL} be the labeled transition system of a HELENALIGHT ensemble specification EnsSpec = $(\Sigma, behaviors)$ together with an admissible initial state such that no role behavior in behaviors starts with a state label. Let T_{PRM} be the labeled transition system of its PROMELALIGHT translation trans(EnsSpec).*
For any $LTL_{\backslash \boldsymbol{X}}$ formula ϕ over AP(EnsSpec), $T_{\text{HEL}} \models \phi \Leftrightarrow T_{\text{PRM}} \models \phi$.

8 Model-Checking HELENALIGHT with Spin

The results from the previous sections allow us to verify LTL properties for a HELENALIGHT ensemble specification by model-checking its PROMELALIGHT translation in Spin. However, the semantics of HELENALIGHT and therefore satisfaction of LTL formulae is defined relatively to a given initial state σ_{init}. Thus, when model-checking the corresponding PROMELALIGHT translation, we have to establish the corresponding initial state $trans_{\mathrm{init}}(\sigma_{init})$ in PROMELALIGHT and verify properties relatively to this initial state. We setup the initial state in a dedicated `init`-process (cf. Fig. 7). To reflect satisfaction of LTL formulae relatively to this initial state, we further extend the original HELENALIGHT LTL formula ϕ to $\Box(init \Rightarrow \phi)$. The *init* is thereby a property which only holds when the initialization in PROMELALIGHT according to the given initial state in HELENALIGHT was finished.

P2P Example: Respecting the aforementioned adaptations to LTL formulae, the goal for our p2p example in Sec. 4.3 is translated to

$$\Box (\ Requester@startRequester \Rightarrow$$
$$\Box(Provider@stateReqFile \Rightarrow \Diamond Requester@stateSndFile) \ ).$$

If we restrict the number of routers, this property holds for the PROMELALIGHT translation in Fig. 7 and we can therefore conclude that it also holds for the HELENALIGHT ensemble specification in Sec. 4.1 for the initial state where only one requester exists. To model-check the more interesting goal from Sec. 3, we have to extend the translation to full HELENA by supporting two additional features, data and components, which adopt roles. In [12], we report the extended translation and argue that stutter trace equivalence holds for this extension as well. Furthermore, we present an automatic code generator based on the XTEXT workbench of Eclipse which takes a HELENA ensemble specification written in HELENATEXT, our domain specific language [13] for HELENA ensembles, as input and generates the PROMELA translation as output.

9 Conclusion

HELENA specifications provide models for dynamically evolving ensembles. This paper deals with a missing link in the HELENA development methodology concerning the early verification of ensemble specifications against goals described by LTL formulae. For this purpose, we proposed to translate HELENA ensemble specifications into PROMELA which can be checked with Spin. To prove the correctness of the translation, we have (a) defined an SOS semantics for simpler variants of HELENA and PROMELA and (b) shown that both are stutter trace equivalent. Hence, LTL formulae (without *next*) are preserved; cf. [1].

Our approach of verification is in-line with goal-oriented requirements approaches like KAOS [17]. They also specify goals by LTL properties. However, they translate their system specifications into the process algebra FSP [18], which is not sufficient to represent the dynamics of ensembles since dynamic

process creation and directed communication are not supported. Techniques for the development of ensembles have been thoroughly studied in the recent AS-CENS project [21]: In [5], ensemble-based systems are described by simplified SCEL programs and translated to PROMELA. However, the translation is neither proved semantically correct nor automated. DFINDER [4] implements efficient strategies exploiting compositional verification of invariants to prove safety properties for BIP ensemble models, but does not deal with dynamic creation of components. DEECo ensemble models [3] are implemented with the Java framework jDEECo and verified with Java Pathfinder [4]. Thus, opposed to HELENA, they do not need any translation. However, since DEECo relies on knowledge exchange rather than message passing, they do not verify communication behaviors. Finally, we would like to point out, that our approach has been strongly inspired by the way how the distributed language KLAIM has been transferred to Maude in [7]. There, the correctness of the translation was established by a stutter bisimulation which preserves CTL^* properties (without *next*). The translation of HELENA into PROMELA is, however, not stutter bisimilar but stutter trace equivalent and thus only preserves LTL formulae (without *next*).

For future work, we plan to conduct more experiments to examine the power of our verification approach. For instance, the question arises how big ensembles can get in terms of role instances to still provide results in reasonable time. For model-checking full HELENA, it is also interesting what impact the topology of the underlying component network (e.g., ring structure, graph structure, etc.) has on the verification of goals for ensemble specifications. As a larger case study, we are currently investigating the power of our verification method for the science cloud platform, a voluntary peer-to-peer cloud computing platform, which was modeled in HELENA in [15].

References

1. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
2. Boronat, A., Knapp, A., Meseguer, J., Wirsing, M.: What Is a Multi-modeling Language? In: Recent Trends in Algebraic Development Techniques. LNCS, vol. 5486, pp. 71–87. Springer (2008)
3. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: The Invariant Refinement Method. In: Software Engineering for Collective Autonomic Systems, LNCS, vol. 8998. Springer (2015)
4. Combaz, J., Bensalem, S., Kofron, J.: Correctness of Service Components and Service Component Ensembles. In: Software Engineering for Collective Autonomic Systems, LNCS, vol. 8998. Springer (2015)
5. De Nicola, R., Lluch-Lafuente, A., Loreti, M., Morichetta, A., Pugliese, R., Senni, V., Tiezzi, F.: Programming and Verifying Component Ensembles. In: From Programs to Systems. LNCS, vol. 8415, pp. 69–83. Springer (2014)
6. Eckhardt, J., Mühlbauer, T., AlTurki, M., Meseguer, J., Wirsing, M.: Stable Availability under Denial of Service Attacks through Formal Patterns. In: Fundamental Approaches to Software Engineering. LNCS, vol. 7212, pp. 78–93. Springer (2012)
7. Eckhardt, J., Mühlbauer, T., Meseguer, J., Wirsing, M.: Semantics, Distributed Implementation, and Formal Analysis of KLAIM models in Maude. Science of Computer Programming 99, 24–74 (2015)

8. Goguen, J.A., Meseguer, J.: Universal Realization, Persistent Interconnection and Implementation of Abstract Modules. In: Proc. of the Colloquium of Automata, Languages and Programming. LNCS, vol. 140, pp. 265–281. Springer (1982)

9. Havelund, K., Larsen, K.G.: The Fork Calculus. In: Proc. of the Colloquium of Automata, Languages and Programming. LNCS, vol. 700, pp. 544–557. Springer (1993)

10. Hennicker, R., Klarl, A.: Foundations for Ensemble Modeling - The Helena Approach. In: Specification, Algebra, and Software. LNCS, vol. 8373, pp. 359–381. Springer (2014)

11. Holzmann, G.: The Spin Model Checker. Addison-Wesley (2003)

12. Klarl, A.: From Helena Ensemble Specifications to Promela Verification Models. Tech. rep., LMU Munich (2015), http://goo.gl/G0sU6U

13. Klarl, A., Cichella, L., Hennicker, R.: From Helena Ensemble Specifications to Executable Code. In: Formal Aspects of Component Software. LNCS, vol. 8997, pp. 183–190. Springer (2015)

14. Klarl, A., Hennicker, R.: Design and Implementation of Dynamically Evolving Ensembles with the Helena Framework. In: Proc. of the Australasian Software Engineering Conference. pp. 15–24. IEEE (2014)

15. Klarl, A., Mayer, P., Hennicker, R.: Helena@Work: Modeling the Science Cloud Platform. In: Leveraging Applications of Formal Methods, Verification and Validation. LNCS, vol. 8802, pp. 99–116. Springer (2014)

16. Lamport, L.: What Good is Temporal Logic? In: IFIP 9th World Congress. pp. 657–668 (1983)

17. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley (2009)

18. Magee, J., Kramer, J.: Concurrency-State Models and Java Programs. Wiley (2006)

19. Meseguer, J., Palomino, M., Martí-Oliet, N.: Algebraic Simulations. Journal of Logic and Algebraic Programming 79(2), 103–143 (2010)

20. Weise, C.: An Incremental Formal Semantics for PROMELA. In: Third SPIN Workshop (1997)

21. Wirsing, M., Hölzl, M., Koch, N., Mayer, P.: Software Engineering for Collective Autonomic Systems, LNCS, vol. 8998. Springer (2015)

22. Wirsing, M., Knapp, A.: A Formal Approach to Object-Oriented Software Engineering. Electrical Notes on Theoretical Computer Science 4, 322–360 (1996)