Energy-Utility Analysis of Probabilistic Systems with Exogenous Coordination*

Christel Baier, Philipp Chrszon, Clemens Dubslaff,
Joachim Klein, and Sascha Klüppelholz

Faculty of Computer Science, Technische Universität Dresden, Dresden, Germany
{christel.baier, philipp.chrszon, clemens.dubslaff,
joachim.klein, sascha.klueppelholz}@tu-dresden.de

Abstract. We present an extension of the popular probabilistic model checker Prism with multi-actions that enables the modeling of complex coordination between stochastic components in an exogenous manner. This is supported by tooling that allows the use of the exogenous coordination language Reo for specifying the coordination glue code. The tool provides an automatic compilation feature for translating a Reo network of channels into Prism's guarded command language. Additionally, the tool supports the translation of reward monitoring components that can be attached to the Reo network to assign rewards or cost to activity within the coordination network. The semantics of the translated model is then based on weighted Markov decision processes that yield the basis, e.g., for a quantitative analysis using Prism. Feasibility of the approach is shown by a quantitative analysis of an energy-aware network system example modeled with a role-based modeling approach in Reo.

1 Introduction

In recent decades, many algorithms, logics and tools have been developed for the formal modeling and analysis of probabilistic systems, combining techniques introduced by the model-checking community with methods for the analysis of stochastic models (see, e.g., [21,15,11]). A widely used model is provided by Markov decision processes (MDPs), which represent probabilistic systems with non-determinism, suitable to model, e.g., concurrency, adversarial behavior or control. To allow for quantitative information attached to the states or transitions, MDPs are often augmented with rewards (sometimes also interpreted as costs). Rewards are useful, e.g,. to reason about energy, waiting times or other costs, as well as utility, such as the number of successful completions of a task. Popular model checkers such as Prism [32,41] or Storm [17] can then be used to

* The authors have been supported by the DFG through the Collaborative Research Center SFB 912 HAEC, the Excellence Initiative by the German Federal and State Governments (cluster of excellence cfAED), the Research Training Group RoSI (GRK 1907), the DFG-projects BA-1679/11-1 and BA-1679/12-1, and the 5G Lab Germany.

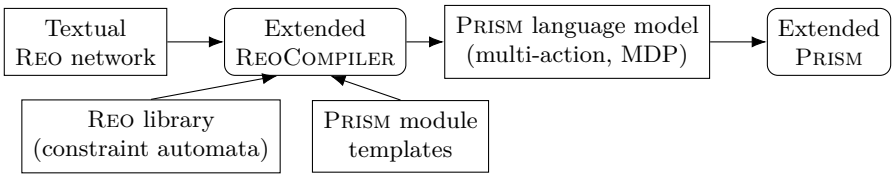

Fig. 1. Using the extended ReoCompiler to generate Prism language models

establish formal guarantees on the expected extremal (maximal/minimal) accumulated rewards and for the analysis of the trade-off between multiple accumulated rewards, e.g., comparing the required energy and the utility gained until reaching a goal for the various ways the non-determinism in the MDP can be resolved (see, e.g., [19,20,9,10]).

For modeling of stochastic systems, a common formalism is the Prism input language, a guarded-command language with probabilistic language features inspired by reactive modules [1]. It allows modeling a system by parallel composition of independent modules that can synchronize over shared actions and is particularly suitable for a symbolic encoding using, e.g., multi-terminal binary decision diagrams (MTBDDs) [41]. However, in practice, modeling complex coordination between the modules can be cumbersome and may require hard-coding the various synchronization possibilities in each module manually. It would therefore be desirable to model the coordination exogenously, i.e., the individual components of the system expose their willingness for synchronization via a well-defined interface to the outside, but do not need to be aware of the concrete connections to the other parts of the system. This facilitates a separation of concerns between computation and coordination, providing modeling flexibility and the ability to easily switch between coordination variants.

A preeminent advocate and example for this exogenous approach is the Reo language [2], a modeling formalism that allows for coordination patterns to be modeled compositionally as a network of channels. There are a wide variety of semantics for Reo [26] and, due to its generality, it can be useful in a wide range of contexts [5,3,8,44,27,30,29]. In the context of (non-probabilistic) model checking of systems described or coordinated by a Reo network, the operational semantics provided by constraint automata [13] proved to be versatile [8,30,29].

Contributions. We present an extension of the Prism input language and provide tool support that permits the use of multi-actions and suitable parallel composition operators that facilitate the exogenous modeling of coordination (Sec. 3). With an underlying MDP-based semantics, the parallel composition operators are derived from a data-abstract variant of *simple probabilistic constraint automata* (spCA) [7]. Here, probabilistic choice can influence the choice of successor state, but does not directly apply to the selection of enabled actions and is thus compatible with the MDP formalism.

Having provided the technical base for exogenous coordination, we are then interested to leverage Reo for the coordination of Prism modules. To achieve this, we have extended the ReoCompiler [46] with support for Prism as a

new target language (Sec. 4). This enables the automatic generation of a PRISM language model description from a textual description of a REO network that coordinates PRISM modules exogenously (see Fig. 1). To attach rewards to activity of the components and the network, we introduce the concept of reward monitors and provide tool support. This allows the quantitative analysis of the performance and of trade-offs for different scheduling and coordination strategies using PRISM's variety of analysis backends (probabilistic model checking using explicit and symbolic engines as well as statistical model checking).

Our main focus is on the use of non-probabilistic REO networks (with a constraint-automata-based operational semantics) for the coordination of probabilistic PRISM modules. However, due to the compatibility with the spCA and MDP semantics, it is also possible to describe and use probabilistic channels by providing their operational behavior in the spCA semantics as PRISM modules and incorporate those into a REO network.

To demonstrate the feasibility of this exogenous modeling approach for the analysis of non-trivial stochastic systems, we consider a case study of a peer-to-peer network with compute nodes that can either play the role of a server, a client, or a relay in the computer network (Sec. 5). For this, we apply the role-based modeling approach using REO as suggested in [16]. Role binding and role playing, as well as the communication protocol for the file transfer is constructed and coordinated via a network of REO channels and connectors. We consider variants where the network topology is replaced and where a particular strategy is employed by switching to a different role-playing coordinator. We demonstrate the analysis of several queries that can be used to illuminate the trade-offs in the strategies. Our extensions of PRISM and the REOCOMPILER, as well as additional material is available at `https://wwwtcs.inf.tu-dresden.de/ALGI/PUB/FA18`.

Related Work. Apart from REO, there is a variety of coordination languages, surveyed, e.g., in [40]. For our case study (Sec. 5), we rely on the models incorporating the concept of roles. Although roles are intuitive and commonly understood, there is no generally accepted definition of roles [47]. We follow the Dresden approach towards roles [31] and rely on our modeling framework for role-based systems using REO presented in [16].

Several approaches extending REO with stochastic component connectors have been presented in the literature, providing semantics in terms of simple probabilistic constraint automata [7], continuous-time constraint automata [14] quantitative intensional automata [4], stochastic REO automata [37], stochastic timed automata for REO [35], and probabilistic timed constraint automata [22], to mention a few. All these approaches above have in common that no direct tool support exists for these models and practical use is mainly justified by providing translations to continuous-time Markov chains (CTMCs) or interactive Markov chains (IMCs) [24]. For instance, case studies have been carried out in [4,37,38], based on IMC and CTMC representations of stochastic REO automata and computing steady-state probabilities using PRISM. In this line, REO2MC, a tool chain to automatically generate CTMC semantics from quantitative in-

tensional automata was presented in [6]. Avoiding intermediate semantics, [39] presented a direct IMC semantics for stochastic Reo and provides tool support using the model checkers CADP and IMCA. Using Prism, they also performed quantitative analysis on CTMCs generated from the IMC semantics, including reward-based properties in the case study of [38].

Concerning modeling formalisms for stochastic systems, there is a variety of other approaches departing from the state-based models such as Markov chains or Markov decision processes we employ in this paper, e.g., stochastic Petri nets [36] or the stochastic process algebra PEPA [25].

2 Preliminaries

In this section we provide a brief overview to the Prism input language, Markov decision processes (MDPs) as underlying semantics and the quantitative measures that can be addressed using probabilistic model checking. For details on Prism we refer, e.g., to [41,42]. Details on MDPs and probabilistic model checking can, e.g., be found in [45,28,12] and the tutorial [18]. In the later sections of the paper we assume the reader to be familiar with the core concepts of Reo. For further details we refer, e.g., to [2,13].

Markov decision processes. A *Markov decision process* (MDP) is a tuple $\mathcal{M} = (S, \mathfrak{Act}, P, \mathit{Rew})$ where S is a finite set of states, $\mathfrak{Act}$ a finite set of actions, $P\colon S \times \mathfrak{Act} \times S \to [0,1] \cap \mathbb{Q}$ is the transition probability function and Rew is a set of reward functions $\mathit{rew}_i\colon S \times \mathfrak{Act} \to \mathbb{N}$. We require that $\sum_{s' \in S} P(s,\alpha,s') \in \{0,1\}$ for all $(s,\alpha) \in S \times \mathfrak{Act}$. We denote by $\mathfrak{Act}(s)$ the set of actions that are enabled in s, i.e., $\alpha \in \mathfrak{Act}(s)$ iff $P(s,\alpha,s') > 0$ for some $s' \in S$. The paths of $\mathcal{M}$ are finite or infinite sequences $s_0\,\alpha_0\,s_1\,\alpha_1\,s_2\,\alpha_2\ldots$ where states and actions alternate such that $P(s_i,\alpha_i,s_{i+1}) > 0$ for all $i \geqslant 0$. Intuitively, in each step first the non-determinism between the enabled actions is resolved and then the successor state is chosen according to the probability distribution. If $\pi{=}s_0\,\alpha_0\,s_1\,\alpha_1\,s_2\,\alpha_2\ldots\alpha_{k-1}\,s_k$ is a finite path, then $\mathit{rew}(\pi){=}\mathit{rew}(s_0,\alpha_0){+}\mathit{rew}(s_1,\alpha_1)+\ldots+\mathit{rew}(s_{k-1},\alpha_{k-1})$ denotes the accumulated reward along π. A *(randomized) scheduler* for $\mathcal{M}$, often also called policy or adversary, is a function σ that assigns to each finite path π a probability distribution over $\mathfrak{Act}(\mathit{last}(\pi))$ resolving the non-determinism in the MDP, where $\mathit{last}(\pi)$ is the last state of π.

The Prism input language. We provide a brief, informal overview of the Prism modeling language (which is also used by other tools and alternative model checkers such as Storm) and its MDP-based semantics. In particular, we concentrate on the features that are used for the synchronization of the individual modules, as our work presented in this paper extends them with features for multi-action synchronization.

A Prism language model description generally consists of a set of *modules* $M_1,\ldots,M_n$. Each module can be seen as an independent process with local state variables, which can be either Boolean or can take values from a fixed integer range. The values of these state variables can only be updated from within the

```
module foo
  s : [0..4] init 0;

  [act1] s = 0 → 1/2 : (s' = 0) + 1/2 : (s' = 1);
  []     s < 4 → 1/4 : (s' = 0) + 3/4 : (s' = s + 1);
endmodule
```

Fig. 2. A simple PRISM module

module, but can be read from other modules. Therefore, state variable names have to be unique across all the modules in the system. In addition to the local variables inside the modules, one can also declare global variables, which can be updated from any module with certain restrictions that ensure the absence of conflicting updates. In addition to modules, PRISM allows defining reward structures that assign costs to either states or transitions.

The global state space of the composed MDP then consists of the Cartesian product of the local variables of all the modules, as well as of the global variables. Thus, each state in the MDP corresponds to a particular variable valuation. The step-wise behavior of a module M_i is specified by a set of *guarded commands* C_i, where each command c_j consists of an action a_j, a state guard g_j and an update specification u_j. The state guard, a Boolean expression over the variable valuations of all variables (global and local in any module), determines whether a command is *locally enabled* in a module. The update specification describes a probability distribution over the updates to the variable valuations. The action of a command allows for the synchronization between modules. A command becomes *globally enabled* only if all synchronization partners provide corresponding locally enabled commands. In standard PRISM, the action consists of an action name or it can be left empty. The latter corresponds to an internal action that can happen at any time the state guard evaluates to `true`. Such actions never synchronize with other actions. Consider the example in Fig. 2. Here, the PRISM module has a single variable `s` with possible values $0, \ldots, 4$ and two guarded commands. The first, with action `act1`, is enabled if variable `s` $= 0$ and, upon execution, will set the value of variable `s` to either 0 or 1, each with probability $1/2$. The second guarded command specifies an internal action, which is enabled as long as `s` < 4 and, upon execution, will reset the value of `s` to 0 with probability $1/4$ or increment the value of `s` by 1 with probability $3/4$. Each module M_i has an action alphabet Act_i, which consists of all the actions that are mentioned in the commands of module M_i.

The composed MDP arises from the parallel composition of the modules $M_1, \ldots, M_n$. PRISM supports several process-algebra operators that allow fine-grained control over the order and synchronization type used in the parallel composition [42,43]. The parallel composition operator $M_1||M_2$, which is used by default, synchronizes commands in M_1 and M_2 that have actions which occur in both action alphabets Act_1 and Act_2. Thus, a command in M_1 with action $a \in Act_1 \cap Act_2$ is only enabled in some state in $M_1||M_2$ if there exists at least one command in M_2 with action a that is enabled as well. In this case, each

enabled a-command in M_1 can be executed with each enabled a-command in M_2. On the other hand, if there is no enabled a-command in M_2, then none of the a-commands in M_1 are enabled. Those commands with actions outside of $Act_1 \cap Act_2$, as well as those without an action, can be executed only by themselves, i.e., in an interleaved manner. In addition to this default parallel composition operator, PRISM supports an operator $M_1|||M_2$, which does not allow any synchronization and instead composes the commands of M_1 and M_2 in an interleaved manner, as well as a composition operator $M_1 \,|Act|\, M_2$ that allows for specifying the set of actions Act over which synchronization happens directly. Thus, $M_1||M_2$ is equivalent to $M_1 \,|Act_1 \cap Act_2|\, M_2$ and $M_1|||M_2$ is equivalent to $M_1 \,|\varnothing|\, M_2$, i.e., using the empty set as the synchronizing alphabet. The action alphabet of the composition of M_1 and M_2 is obtained as the union of Act_1 and Act_2. Additionally, there is an operator that supports *hiding* of actions, i.e., turning some named actions into internal, empty actions and removing the actions from the action alphabet, as well as an operator for *renaming* actions.

Quantitative analysis. Probabilistic model checkers such as PRISM and STORM can be used for the automated analysis of MDPs, for example answering questions such as "What is the maximal (minimal) probability for reaching some goal state, ranging over all schedulers?". Observing the rewards in the MDP, which can for example be used to model costs, energy, utility, etc., such tools also support a reward-based analysis, e.g., computing the maximal (minimal) expected accumulated reward until some goal is reached. Here, a trade-off analysis between multiple reward functions is of particular interest, for example using multi-objective analysis [19,20] or analysis of an energy-utility trade-off [9,10].

3 Exogenous coordination with Prism

We have extended PRISM's guarded command language with features that facilitate the modeling of more complex coordination schemes, in particular exogenous coordination. Most importantly, we have conservatively extended the PRISM language to support multi-actions. Although multi-actions arise rather naturally in REO connectors coordinating the activity and communication of components, till now there has been no support for in PRISM.

Extending the Prism language with multi-actions. A command in our extension comprises a (possibly empty) *set* of actions α, a state guard, and an update specification. The actions α can either occur in a closed form, denoted by $[\alpha]$ or an open form, denoted by $]\alpha[$. Intuitively, a closed multi-action indicates that no further action can be added during composition and yield a multi-action $\alpha' \subseteq \alpha$, while an open multi-action allows the composition with other actions to form a multi-action $\alpha' \supseteq \alpha$. Note that this extension is conservative in the sense that if α occurs only in closed form and contains at most one action, every command is as in standard PRISM. As before, the action alphabet Act_i of module M_i is obtained from the set of actions that occur in any of M_i's commands.

Using the well-known SOS notation, we now provide the rules for the $M_1|Act|M_2$ parallel composition operator (see Fig. 3) that supports multi-actions. As noted

$$\text{(1)}\quad \frac{[\alpha_1] : g_1 \to u_1 \in C_1 \quad \wedge \quad [\alpha_2] : g_2 \to u_2 \in C_2 \quad \wedge \quad \alpha_1 = \alpha_2 \ \wedge \ \alpha_1 \cap Act \neq \varnothing}{[\alpha_1 \cup \alpha_2] : g_1 \wedge g_2 \to u_1 \cdot u_2 \in C_{1||2}}$$

$$\text{(2a)}\quad \frac{[\alpha_1] : g_1 \to u_1 \in C_1 \ \wedge \ \alpha_1 \cap Act = \varnothing}{[\alpha_1] : g_1 \to u_1 \in C_{1||2}} \qquad \text{(2b)}\quad \frac{[\alpha_2] : g_2 \to u_2 \in C_2 \ \wedge \ \alpha_2 \cap Act = \varnothing}{[\alpha_2] : g_2 \to u_2 \in C_{1||2}}$$

$$\text{(3)}\quad \frac{]\alpha_1[: g_1 \to u_1 \in C_1 \quad \wedge \quad]\alpha_2[: g_2 \to u_2 \in C_2 \quad \wedge \quad \alpha_1 \cap Act = \alpha_2 \cap Act}{]\alpha_1 \cup \alpha_2[: g_1 \wedge g_2 \to u_1 \cdot u_2 \in C_{1||2}}$$

$$\text{(4a)}\quad \frac{]\alpha_1[: g_1 \to u_1 \in C_1 \ \wedge \ \alpha_1 \cap Act = \varnothing}{]\alpha_1[: g_1 \to u_1 \in C_{1||2}} \qquad \text{(4b)}\quad \frac{]\alpha_2[: g_2 \to u_2 \in C_2 \ \wedge \ \alpha_2 \cap Act = \varnothing}{]\alpha_2[: g_2 \to u_2 \in C_{1||2}}$$

$$\text{(5a)}\quad \frac{]\alpha_1[: g_1 \to u_1 \in C_1 \ \wedge \ [\alpha_2] : g_2 \to u_2 \in C_2 \ \wedge \ \alpha_2 \neq \varnothing \ \wedge \ \alpha_1 = \alpha_2 \cap Act}{[\alpha_1 \cup \alpha_2] : g_1 \wedge g_2 \to u_1 \cdot u_2 \in C_{1||2}}$$

$$\text{(5b)}\quad \frac{[\alpha_1] : g_1 \to u_1 \in C_1 \ \wedge \ ]\alpha_2[: g_2 \to u_2 \in C_2 \ \wedge \ \alpha_1 \neq \varnothing \ \wedge \ \alpha_2 = \alpha_1 \cap Act}{[\alpha_1 \cup \alpha_2] : g_1 \wedge g_2 \to u_1 \cdot u_2 \in C_{1||2}}$$

Fig. 3. SOS rules for the parallel composition of the commands of two modules, synchronizing over the action alphabet Act.

above, $M_1||M_2$ can be obtained by using $Act = Act_1 \cap Act_2$ as the synchronization alphabet. In Fig. 3, we denote by $[\alpha] : g \to u \in C_i$ that there is a command in module M_i with closed multi-action α, state guard g and update specification u. Similarly, $]\alpha[: g \to u \in C_i$ denotes the same command albeit with open multi-action α. In the bottom part of the rules, $C_{1||2}$ stands for the commands in the composed module $M_1||M_2$. Furthermore, $u_1 \cdot u_2$ stands for the combined update specification obtained from u_1 and u_2 by using their product distribution, just as in the standard PRISM semantics. For instance, the combined update specification $u_1 \cdot u_2$ for $u_1 = 1/2 : (s'{=}0) + 1/2 : (s'{=}1)$ and $u_2 = 1/3 : (t'{=}0) + 2/3 : (t'{=}1)$ would be

$$1/6 : (s'{=}0, t'{=}0) + 2/6 : (s'{=}1, t'{=}0) + 1/6 : (s'{=}0, t'{=}1) + 2/6 : (s'{=}1, t'{=}1).$$

We now provide some intuitive explanations for the composition rules. Rule (1) concerns the synchronization of two commands with closed multi-actions. As both are closed, it is not possible to add additional actions, which implies that $\alpha_1 = \alpha_2 = \alpha_1 \cup \alpha_2$. The condition $\alpha_1 \cap Act \neq \varnothing$ ensures that there is at least one action available for synchronization. All commands with closed actions that do not have any synchronizing action are handled by the symmetrical rules (2a) and (2b). This includes the handling of the closed empty multi-action, clearly excluded from the scope of rule (1). Altogether, rules (1), (2a) and (2b) collapse to the standard composition operator of PRISM whenever the multi-actions are singletons or empty, thus preserving the standard PRISM semantics whenever neither multi-actions nor open actions are used.

Rules (3), (4a) and (4b) deal with the composition of commands with open multi-actions. Rule (3) allows the parallel execution of two commands whenever their actions agree on the synchronized action alphabet. Note that there is no restriction on the non-emptiness of $\alpha_1 \cap Act$ and $\alpha_2 \cap Act$. Thus, two open commands that do not have actions in the synchronization alphabet and are

therefore "unrelated" can be executed in parallel. Likewise, by rules (4a) and (4b), those actions can also be executed without synchronization.

Rules (5a) and (5b) deal with the parallel composition of open and closed commands. For (5a), the condition $\alpha_2 \neq \varnothing$ ensures that closed, empty actions never synchronize, while $\alpha_1 = \alpha_2 \cap Act$ ensures that $\alpha_1 \subseteq \alpha_2$, i.e., α_1 introduces no new actions, and that α_1 agrees with α_2 on the synchronizing actions. Rule (5b) is the symmetric rule to rule (5a).

The three rules (3), (4a) and (4b) correspond to the product rules for a data-abstract *simple probabilistic constraint automaton* as presented in [7]. The other rules in Fig. 3 can be similarly seen as variants of those product rules, adapted for closed commands and the mixture of closed and open commands in a natural, backward compatible fashion. Note that our parallel composition is commutative and associative, i.e., for modules M_1, M_2, and M_3 we have that the semantics of $M_1 \parallel M_2$ is isomorphic to the semantics of $M_2 \parallel M_1$ and likewise, the semantics of $M_1 \parallel (M_2 \parallel M_3)$ is isomorphic to the semantics of $(M_1 \parallel M_2) \parallel M_3$. The proof of this statement is straightforward but tedious and is provided in the appendix.

In the translation from the PRISM language model description to the underlying MDP, the set of actions in the MDP then corresponds to the powerset of action names that appear in the model description. That is, for an action alphabet Act of the composed system, the set of actions in the MDP is then $\mathfrak{Act} = 2^{Act}$, i.e., each action in the MDP is a subset of Act.

Reward structures in PRISM can be used to assign rewards to state-action pairs in the MDP, by declaring reward values for states satisfying a state guard and a specific, single action name. We have extended the declarations of reward structures with support for multi-action specifications, i.e., of the form $[\alpha]$ and $]\alpha[$ in the definitions of reward structures, where α is a set of actions from the action alphabet Act of the composed system. The reward value is then assigned to state-action pairs in the MDP with matching actions. A specification $[\alpha]$ matches exactly the action $\beta \in \mathfrak{Act}$ in the MDP iff $\alpha = \beta$. For $]\alpha[$, all actions $\beta \in \mathfrak{Act}$ that satisfy $\alpha \subseteq \beta$ match and are assigned the reward value. This can be used, e.g., to assign a reward whenever a particular action name is active, irregardless of which other actions in the system are active simultaneously.

Further extensions of the Prism language. We have extended the PRISM language with additional features that simplify exogenous and compositional model design. PRISM supports a mechanism to take one module and obtain an additional instance. As the variable names and action names of a module live in the same namespace (even local variables of a module can be read from other modules), this requires renaming all module variables and possibly the action names in case synchronization between instances should be avoided. For example, the PRISM statement

$$\texttt{module M2} \;=\; \texttt{M1 [s1} = \texttt{t1}, \texttt{s1} = \texttt{t2}, \texttt{a1} = \texttt{a2}\texttt{] endmodule}$$

constructs module `M2` as a copy of `M1`, renaming the state variables `s1` and `s2` to `t1` and `t2`, respectively, as well as renaming action `a1` to `a2`. As this kind of statement requires detailed knowledge of the variable names for the module,

we have extended the syntax to allow for *rule-based renaming*. For example, the statement

$$\texttt{module M2} \;=\; \texttt{M1 (varprefix = M1\_)[a1 = a2] endmodule}$$

would automatically rename a variable s in `M1` to `M1_s`, which makes it easy to ensure global uniqueness of variable names. Additionally, we support rules with `varsuffix` (adding a given suffix to the variable names), `actionprefix` and `actionsuffix` (similarly renaming the individual action names occurring in a module). Rule-based renaming is being performed first, then the additional, explicitly given, renamings are performed. Note that, however, every state variable and action can only be renamed once. With this automatic renaming, PRISM's module renaming statement can be seen as the *instantiation* of a module. However, in standard PRISM, every module definition that appears in the input file is automatically instantiated. This makes it impossible to provide a library of module templates, of which only a subset is actually instantiated. To remedy this issue, we allow a module definition to be marked as *template*. Such a template module will not be instantiated automatically, but is available for instantiation via module renaming.

A simple example of exogenous coordination. As an example for exogenous coordination, consider a simple setting with three producer modules and three consumer modules. Each of them has a certain probability in each step to become broken. In each step, exactly one of the non-broken producers shall synchronize with one of the non-broken consumers, until eventually almost surely all have failed. In the standard PRISM language, we have to hard-code one command for each synchronization choice in each module, e.g., by using actions $\texttt{p}_\texttt{i}\texttt{c}_\texttt{j}$ to synchronize producer i with consumer j. With our extension of PRISM, we can model exogenous coordination: Each producer and consumer module has a single action, which is suitably synchronized by some glue code modules, e.g., a merger module that nondeterministically selects one of the producers and is chained to a router module that nondeterministically selects one of the consumers. In Appendix B, we provide a detailed description of both approaches. It is readily apparent that the second, exogenous approach provides far greater flexibility and separation of concerns, making it easy to replace the coordination glue code by alternative variants, specializations, etc.

Prism implementation. We have extended PRISM with support for handling multi-actions and for dealing with the other proposed language extensions, both in PRISM's explicit engine (where an explicit, graph-based model representation is built) and in the (semi-)symbolic engines (where a symbolic model representation [41] is used). For the explicit engine, this mostly consists of the handling of the parallel composition according to the rules of Fig. 3 during model construction. For a given variable valuation, we can easily determine the commands that are locally enabled in each module of the system. Then, we have to determine the possibilities for synchronized and independent execution of commands from different modules, maintaining data structures to speed-up the lookup of potential synchronizing commands.

For the symbolic engine of Prism, which is based on a symbolic representation of the model via *multi-terminal binary decision diagrams* (MTBDDs) [41], the transition structure of an MDP is encoded using MTBDD variables for the nondeterministic choices, as well as variables for encoding the states and successor states, mapping to the probability for a given transition in the MDP. As Prism already encodes each action name in the model description by one variable, adapting the encoding to multi-actions is rather straightforward. Likewise, the various composition rules in Fig. 3 can be elegantly formulated as symbolic operations on the MTBDD representation for each module. One complication however is the encoding of local nondeterminism within a module, i.e., to distinguish which of multiple commands with the same multi-action that are enabled simultaneously are actually executed. The encoding used by standard Prism (a binary encoding of an integer for the various local nondeterministic choices) is not convenient for a fully symbolic composition, therefore we changed this encoding. In our extension, each command in the model corresponds to an MTBDD variable that denotes whether this command is actually active or inactive in a given step.

4 Reo for exogenous coordination within Prism

Having extended Prism by the infrastructure for the exogenous coordination of probabilistic components, we are now interested in a framework for the convenient modeling of the coordination glue code. For this, the channel-based coordination language Reo [2] provides an elegant and compositional modeling approach, where the coordination glue code for components is specified using a Reo network of channels that can be used to model a plethora of coordination patterns. For this, both stateless channels such as synchronous channels (ensuring that activity at their channel ends happens simultaneously), asynchronous channels (ensuring the non-synchronicity/mutual exclusion at their channel ends), lossy channels or transformer channels, as well as stateful ones such as FIFO channels (which can accept a token or data and pass it on later) are used, mediated by network nodes that coordinate the activity of the connected channels. Additionally, ready-made or user-defined circuits can be used as building blocks to model common coordination patterns, such as a sequencer that ensures that certain activity happens one after the other. With constraint-automata-based operational semantics [13,7] for Reo, the behavior of the whole network can be obtained from the automata-based descriptions of the individual parts (channels and nodes) in a compositional manner by a series of product operations.

To allow the use of Reo as the coordination glue code of Prism components, we make use of the ReoCompiler tool developed at the Centrum Wiskunde & Informatica, Amsterdam [46]. Among others, the ReoCompiler supports the convenient textual specification of Reo networks, providing the glue code for components. Then, it allows the compilation of the glue code to a target language (such as Java). When combined with definitions of the components in the target language (e.g., a Java class implementing the component's behavior),

the coordinated system can then be executed. The external components interface with the Reo network via input and output ports.

Prism as a target language of the ReoCompiler. We have extended the existing ReoCompiler with support for the Prism language. In particular, we provide a translation from the constraint-automata-like intermediate compilation result for the glue code to the Prism language. This relies on the extension for multi-actions and the product operator for modules presented in Sec. 3, which allows the encoding of the operational semantics via (data-abstract) simple probabilistic constraint automata [7]. Together with the flexible module instantiation from module templates, the generated Prism language model description properly instantiates the various (Prism-based) components and connects with the generated coordination glue code. Here, Prism's components actions are exported to the network as input/output ports.

The constraint-automata semantics for the Reo channels supports the transfer of data, i.e., ports or nodes in the network are not only active or not, but may have some observable data value. As we are mainly interested in the data-abstract coordination of Prism components, i.e., an action either fires or not but carries no data, we treat the Reo network as using a singleton data domain. As sometimes attaching data to actions is natural for certain modeling tasks, we however provide basic tooling as well to emulate actions carrying data by encoding the different data values as variants of the actions, i.e., for an action $\mathtt{a}$ there are variants $\mathtt{a}_1$, $\mathtt{a}_2$, ... corresponding to the data values $1, 2, \ldots$.

We support two orthogonal approaches to the compilation. In the first, *monolithic* approach, the whole Reo network comprising the glue code, i.e., all parts of the network except for the "native" Prism components are compiled into a single protocol module. This compilation relies on the composition of all the channels and nodes within the ReoCompiler. In a second, *compositional* approach, the ReoCompiler is used to generate a Prism language file where all the individual channels and nodes of the Reo network are translated to individual Prism modules and where the composition of the behavior is performed during Prism's translation from the model description to the concrete MDP. Here, we crucially rely on the fact that we can readily translate the ReoCompiler's internal representation of Reo networks into a Prism module. It should be noted that both approaches have a minor difference in the underlying semantics: The composition inside the ReoCompiler relies on classical interleaving for independent (unsynchronized) parts of the Reo network. Then, the generated code realizes a sequential implementation that simulates the parallel execution of these independent parts. In the compositional approach, unrelated actions can synchronize (cf. rule (3) in Fig. 3, with $Act = \varnothing$) and thus are executed in parallel. This can, e.g., be observed for chains of FIFO channels. The monolithic approach can be useful to hide the internal complexity of a Reo network from Prism, while the compositional approach provides more insight into the parts and the structure of the Reo network at the level of the model checker.

Further extensions to the ReoCompiler. We have extended the ReoCompiler with some other features that are useful in the context of the quantitative

analysis of the generated models. First, towards model checking it is often required to refer to the content of a state variable, e.g., for one of the PRISM components or the state of a FIFO channel in the REO network. As the focus of the REOCOMPILER is more on generating executable code where the component names are largely irrelevant, it only generates unique, but not necessarily stable names. We have added syntax and support for providing a name during component instantiation, which results in a predictable name for the state variables of the component instances (and memory cells for stateful channels) in the generated PRISM language file.

Another common requirement in probabilistic model checking is the ability to assign *rewards* or *costs* to the model, for example to model the energy consumption or to track the achieved utility on completion of a task. In PRISM, such rewards can be attached to states (e.g., for every step spent in a state, a certain reward is accumulated) as well to transitions (e.g., for a step with certain actions, a given reward is accumulated). As the names of the actions (i.e., ports and nodes) generated during the network composition process are not necessarily stable or predictable, e.g., due to the application of the REO hiding operator, we have extended REOCOMPILER with support for *reward monitors*. Here, a reward monitor is a special component with a given set of input ports which can be attached to the network using the standard REO channels and operations. The reward-monitor definition then specifies the rewards that are assigned whenever certain of the input ports of the monitor are active. We support two variants, local and global monitors. A local monitor tracks a reward on its own, resulting in a single reward structure in the generated PRISM file. In contrast, a global monitor carries a label that ensures that, if there are multiple monitors with the same label, the reward from all of those monitors is collected in a single PRISM reward structure. This allows, e.g., attaching a dedicated reward monitor to each component that records the energy consumption when there is port activity, with all those rewards being added together to yield the overall energy consumption of the system in each step.

Additionally, we have added the ability to include PRISM language snippets from external files into the generated PRISM language file, allowing the convenient inclusion of the module templates for the PRISM components that may be instantiated in the generated model description, as well as auxiliary definitions that commonly arise during the modeling with PRISM, such as constant definitions, the definition of state labels as well as additional reward structures.

Example. In Appendix B.3, we provide a detailed description how the coordination glue code in the producer/consumer example can be elegantly modeled using a REO network, with automatic generation of the corresponding PRISM language model description via our extended version of the REOCOMPILER. Here, we can model the coordination using REO channels and an exclusive router, which provides the desired coordination in a compositional manner. For an illustration of the use of reward monitors, we refer to Appendix B.4.

Other model semantics. In addition to the MDP semantics, it is also possible to generate a model description for a discrete-time Markov chain (DTMC). Here,

PRISM resolves all nondeterminism in the MDP model uniformly. Moreover, it is possible as well to select probabilistic timed automata (PTA) semantics [34], where the PRISM modules may additionally contain special clock variables, clock invariants and the commands can contain clock guards and clock resets. For these PTA models, our extension for multi-actions and the related composition operators supports the analysis via PRISM's digital clock engine, which internally transforms the PTA into an MDP [33]. The adaption of PRISM's other PTA analysis engines remains part of future work.

5 Application: Energy-aware Network System

In this section, we present a peer-to-peer file transfer case study that leverages the extensions to PRISM and the REO tool support to model the complex coordination between the components of the system. The model is inspired by a case study presented in [23]. The network system consists of several *stations* or nodes interconnected via a network with some fixed topology, e.g. a ring or star topology. Each of the stations can store files. We do not consider the actual contents of these files in our model and rather represent them using an abstract index. A station may request a file from another station connected to the network. Then, the file is transfered between the stations using a peer-to-peer approach, i.e., without a central entity handling the transfer.

In a file transfer, each participating station may act in one of three different *roles*. The station that initiated the request and will receive the file plays the role of the *client*. Conversely, a station that has a local copy of the requested file can act as a *server*. Since a file transfer can also happen between stations that are not directly connected, but via one or more hops, the stations in between client and server play the role of a *relay*. A relay station retransmits incoming requests and file data to its neighboring stations. As a file transfer may be initiated between any two stations on the network, each station may dynamically play one of the three roles: server, client or relay.

To model such a system, we employed the role-based modeling approach proposed in [16]. Within this approach, the dynamically changing behaviors, i.e., the roles, are separated from the static core functionality. The main idea is to encapsulate the role behaviors into *role components*. These role components are then bound to their player using a REO connector. This *binding connector* enables the player component to dynamically enact the role behavior.

Figure 4 depicts the binding connector between a station and its roles in detail. Here, $\bullet$ denotes standard REO nodes with nondeterministic merging on the input side and replication (simultaneous activity) on the output side, while $\otimes$ denotes an x-router node, where there is a nondeterministic choice on the output side. The station component as well as the role components are modeled as PRISM modules. Each of the role components is wrapped in a role adapter (shown in detail for the client role). This adapter adds one port that allows enabling or disabling the role. Internally, this port is synchronized with the `in` and `out` ports of the role component, thus blocking the `act` port will effectively

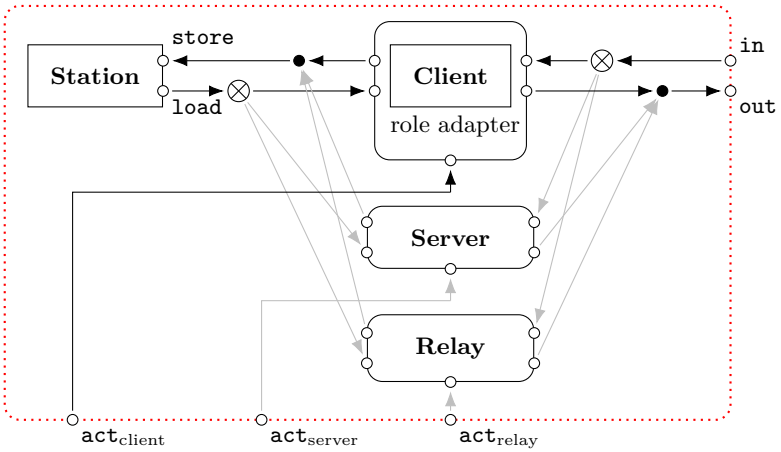

Fig. 4. A station component and its bound role components

disable the role. The connector between the station component and the role components ensures that each role can retrieve the file stored on the local station or replace it with another one. The other part of the connector attaches the roles to the network, allowing each of them to send or receive requests or file data. We use the same binding connector for all stations within the network. The network itself is also realized as a Reo network which connects the `in` and `out` ports of the stations according to the network topology.

The `act` ports of a station's roles allow the *role-playing coordinator* to enable and disable role behaviors dynamically. The role-playing coordinator enforces that all stations act according to the peer-to-peer file-transfer protocol. The core component of a station can generate a request for a certain file according to a probabilistic distribution. This request is buffered by the coordinator. Eventually, the coordinator allows the station to play the client role to send the request into the network. Another station will then receive this request. In case this station has the requested file, the server role will be enabled, which in turn fetches the file and sends it back. If the station does not have the file, the relay role will be played and the request is sent to the neighboring stations. For simplicity, the global coordination will also ensure that only one file transfer happens simultaneously.

Our approach allows us to vary the coordination without modifying the Prism modules "implementing" the station core component and the role components. The connector modeling the network can be changed to different topologies. The remaining nondeterminism in the system stands for the different strategies that may be employed to achieve certain objectives. This concerns, e.g., the role-playing assignments for the different stations and the choice which of the pending requests will be processed next. By attaching different coordination, we can thus explore the effect of a particular strategy, for example replacing the nondeterministic choice of the next request by a uniform random choice. The coordinator could also be augmented to ensure that no file is "forgotten" by the network by blocking requests that would overwrite the last copy of a file.

We have analyzed the file-transfer model with three stations. In particular, we have considered four variants of the model by using a ring or chain topology of the network connector and by either using a nondeterministic choice or a probabilistic choice of the next request to be processed.[1] For the analysis, we have added reward monitors and state rewards to the model. *Energy* is consumed on network activity, i.e., whenever one or more of the `in` and `out` ports of the network connector are active. Furthermore, a *penalty* (negative utility) is associated with pending requests that have not yet been processed. We have used Prism to analyse the model variants, among others asking for

(a) the minimal/maximal probability that eventually station 1 receives its requested file,
(b) the minimal/maximal probability that eventually all stations have a file,
(c) the minimal expected time until the file requested by station 1 is delivered,
(d) the minimal expected time until all stations have received a file,
(e) the maximal probability to deliver a file to station 1 with less than x penalty,
(f) the maximal probability for delivering a file using a given energy budget without overstepping the penalty threshold, and
(g) the minimal energy required such that a file is delivered to station 1 with a probability greater than 0.9 without a penalty violation.

The analysis results for the queries (c) to (g) are presented in Table 1. The results for (c) show that in a ring topology, the requested file is delivered faster than in the chain topology. This is as expected, since in the chain topology, we always need one hop to transfer a file between the two outer stations of the network, while in the ring topology a direct transfer without hops is always possible. The same argument also applies to (d). The results for (e) show the difference between the random scheduling and the optimal scheduling of the next file transfer. Generally, the random scheduling collects a higher penalty, which means that pending transfers are kept waiting longer. The reward-bounded reachability probability (f) and the quantile [9] query (g) illuminate the trade-off between early processing of a request and thus consuming less energy, or waiting for another request to arrive thereby collecting a penalty for pending requests. Comparing the minimal energy consumption in (g) for the nondeterministic and random selection of the requests, we see that the nondeterministic choice uses less energy. This is as expected, because the nondeterministic selection corresponds to the optimal strategy for choosing the next request.

Model sizes and the time required for model construction and analysis of instances of queries (f) as well as (g) are presented in Table 2. The number of components consists of the number of channels, Prism modules and Reo nodes in the network. The actions column refers to the number of unique action names within the generated model. Here, the analysis has been carried out using the symbolic engine of Prism and the monolithic approach. The considerable number of components and states is caused by the detailed modeling of

[1] For further details on the models and experiments, see `https://wwwtcs.inf.tu-dresden.de/ALGI/PUB/FA18`.

Table 1. Analysis results for the file-transfer model with 3 stations

Variant	(c)	(d)	(e)	(f)	(g)
chain, nondet	4.0	16.15	0.95	0.98	10.0
ring, nondet	2.0	15.94	1.0	0.96	12.0
chain, random	5.87	18.73	0.64	0.65	15.0
ring, random	4.0	18.34	0.78	0.72	12.0

Table 2. Model sizes and analysis times for the file-transfer model with 3 stations

				Time (s)		
Variant	States	Components	Actions	Build	Analysis (f)	Analysis (g)
chain, nondet	4204	108	150	10.7	81.1	58.4
ring, nondet	34164	112	150	90.7	197.8	91.3
chain, random	12612	103	154	10.6	92.0	24.4
ring, random	102492	107	154	62.6	224.5	101.0

the role-playing coordinator. The coordinator divides a file transfer into multiple steps which requires storing request messages in its internal state. The number of states within the random variants is greater than in the nondeterministic variants because the random selection of requests requires additional internal state compared to a nondeterministic selection. The ring topology further increases the number of states since more routes within the network are possible.

6 Conclusions

We have extended the Prism language and the Prism model checker by features that allow an exogenous modeling of the coordination of Prism modules. We believe that, already on its own, these modeling capabilities will be very useful for the modeling of complex case studies. By using our extension of the ReoCompiler, this exogenous approach can additionally leverage the elegant specification of complex coordination patterns by Reo networks and allow the creation of model variants, as seen in our case study.

As future work, we are interested in exploring the full integration of actions with attached data values into Prism. Previous experience with the symbolic encoding of models with data [8] suggest that this would require some effort to ensure a compact symbolic encoding, which is compounded by the fact that good heuristics for the variable ordering from the non-probabilistic setting, such as interleaving the data on the actions with related state variables, may conflict with variable-ordering restrictions designed for efficient probabilistic model checking.

We are also interested in ways to provide the user more feedback during modeling, e.g., by integrating a visualization of the Reo network with animated control flow into Prism's simulation view, which can also be used to explore counter-examples from Prism's non-probabilistic CTL and LTL checkers.

References

1. R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.
2. F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14:329–366, 2004.
3. F. Arbab, L. Astefanoaei, F. S. de Boer, M. Dastani, J. C. Meyer, and N. A. M. Tinnemeier. Reo connectors as coordination artifacts in 2APL systems. In *Intelligent Agents and Multi-Agent Systems (PRIMA'08)*, volume 5357 of *LNCS*, pages 42–53. Springer, 2008.
4. F. Arbab, T. Chothia, R. van der Mei, S. Meng, Y. Moon, and C. Verhoef. From coordination to stochastic models of QoS. In *Coordination Models and Languages (COORDINATION'09)*, volume 5521 of *LNCS*, pages 268–287. Springer, 2009.
5. F. Arbab, N. Kokash, and S. Meng. Towards using Reo for compliance-aware business process modeling. In *Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'08)*, volume 17 of *Communications in Computer and Information Science*, pages 108–123. Springer, 2008.
6. F. Arbab, S. Meng, Y. Moon, M. Z. Kwiatkowska, and H. Qu. Reo2MC: a tool chain for performance analysis of coordination models. In *ESEC/SIGSOFT FSE'09*, pages 287–288. ACM, 2009.
7. C. Baier. Probabilistic models for Reo connector circuits. *Journal of Universal Computer Science*, 11(10):1718–1748, October 2005.
8. C. Baier, T. Blechmann, J. Klein, and S. Klüppelholz. Formal verification for components and connectors. In *Formal Methods for Components and Objects (FMCO'08)*, volume 5751 of *LNCS*, pages 82–101. Springer, 2009.
9. C. Baier, M. Daum, C. Dubslaff, J. Klein, and S. Klüppelholz. Energy-utility quantiles. In *NASA Formal Methods (NFM'14)*, volume 8430 of *LNCS*, pages 285–299. Springer, 2014.
10. C. Baier, C. Dubslaff, J. Klein, S. Klüppelholz, and S. Wunderlich. Probabilistic model checking for energy-utility analysis. In *Horizons of the Mind. A Tribute to Prakash Panangaden*, volume 8464 of *LNCS*, pages 96–123. Springer, 2014.
11. C. Baier, B. R. Haverkort, H. Hermanns, and J. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, 2003.
12. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
13. C. Baier, M. Sirjani, F. Arbab, and J. J. M. M. Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61(2):75 – 113, 2006.
14. C. Baier and V. Wolf. Stochastic reasoning about channel-based component connectors. In *Coordination Models and Languages (COORDINATION'06)*, volume 4038 of *LNCS*, pages 1–15. Springer, 2006.
15. A. Bianco and L. de Alfaro. Model checking of probabilistic and non-deterministic systems. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS'95)*, volume 1026 of *LNCS*, pages 499–513, 1995.
16. P. Chrszon, C. Dubslaff, C. Baier, J. Klein, and S. Klüppelholz. Modeling role-based systems with exogenous coordination. In *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, volume 9660 of *LNCS*, pages 122–139. Springer, 2016.
17. C. Dehnert, S. Junges, J. Katoen, and M. Volk. A Storm is coming: A modern probabilistic model checker. In *Computer Aided Verification (CAV'17), Part II*, volume 10427 of *LNCS*, pages 592–600. Springer, 2017.

18. V. Forejt, M. Z. Kwiatkowska, G. Norman, and D. Parker. Automated verification techniques for probabilistic systems. In *School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM'11)*, volume 6659 of *LNCS*, pages 53–113. Springer, 2011.
19. V. Forejt, M. Z. Kwiatkowska, G. Norman, D. Parker, and H. Qu. Quantitative multi-objective verification for probabilistic systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11)*, volume 6605 of *LNCS*, pages 112–127. Springer, 2011.
20. V. Forejt, M. Z. Kwiatkowska, and D. Parker. Pareto curves for probabilistic model checking. In *Automated Technology for Verification and Analysis (ATVA'12)*, volume 7561 of *LNCS*, pages 317–332. Springer, 2012.
21. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:512–535, 1994.
22. K. He, H. Hermanns, and Y. Chen. Models of connected things: On priced probabilistic timed Reo. In *41st IEEE Annual Computer Software and Applications Conference (COMPSAC'17), Volume 1*, pages 234–243, 2017.
23. R. Hennicker and A. Klarl. Foundations for ensemble modeling – the Helena approach. In *Specification, Algebra, and Software*, volume 8373 of *LNCS*, pages 359–381. Springer, 2014.
24. H. Hermanns. *Interactive Markov chains.* PhD thesis, University of Erlangen-Nuremberg, Germany, 1999.
25. J. Hillston. *A compositional approach to performance modelling.* PhD thesis, University of Edinburgh, UK, 1994.
26. S. T. Q. Jongmans and F. Arbab. Overview of thirty semantic formalisms for Reo. *Scientific Annals of Computer Science*, 22(1):201–251, 2012.
27. S. T. Q. Jongmans, F. Santini, M. Sargolzaei, F. Arbab, and H. Afsarmanesh. Automatic code generation for the orchestration of web services with Reo. In *Service-Oriented and Cloud Computing (ESOCC'12)*, volume 7592 of *LNCS*, pages 1–16. Springer, 2012.
28. L. Kallenberg. *Markov Decision Processes.* Lecture Notes. University of Leiden, 2011.
29. N. Kokash and F. Arbab. Formal design and verification of long-running transactions with extensible coordination tools. *IEEE Transactions on Services Computing*, 6(2):186–200, 2013.
30. N. Kokash, C. Krause, and E. de Vink. Reo + mCRL2: A framework for model-checking dataflow in service compositions. *Formal Aspects of Computing*, 24(2):187–216, 2012.
31. T. Kühn, M. Leuthäuser, S. Götz, C. Seidl, and U. Aßmann. A metamodel family for role-based modeling and programming languages. In *Software Language Engineering (SLE'14)*, volume 8706 of *LNCS*, pages 141–160. Springer, 2014.
32. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.
33. M. Z. Kwiatkowska, G. Norman, D. Parker, and J. Sproston. Performance analysis of probabilistic timed automata using digital clocks. *Formal Methods in System Design*, 29(1):33–78, 2006.
34. M. Z. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theoretical Computer Science*, 282(1):101–150, 2002.

35. Y. Li, X. Zhang, Y. Ji, and M. Sun. Capturing stochastic and real-time behavior in Reo connectors. In *Symposium on Formal Methods: Foundations and Applications (SBMF'17)*, volume 10623 of *LNCS*, pages 287–304. Springer, 2017.
36. M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. Modelling with generalized stochastic Petri nets. *SIGMETRICS Performance Evaluation Review*, 26(2):2, 1998.
37. Y. Moon, A. Silva, C. Krause, and F. Arbab. A compositional model to reason about end-to-end QoS in stochastic Reo connectors. *Science of Computer Programming*, 80:3–24, 2014.
38. Y.-J. Moon, F. Arbab, A. Silva, A. Stam, and C. Verhoef. Stochastic Reo: a case study. In *Workshop on Harnessing Theories for Tool Support in Software (TTSS'11)*, pages 90–105, 2011.
39. N. Oliveira, A. Silva, and L. S. Barbosa. Imc_{reo}: Interactive Markov chains for stochastic Reo. *J. Internet Serv. Inf. Secur.*, 5(1):3–28, 2015.
40. G. A. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*, 46:329 – 400, 1998.
41. D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002.
42. PRISM Manual. `http://www.prismmodelchecker.org/manual/`.
43. PRISM Language Semantics. `http://www.prismmodelchecker.org/doc/semantics.pdf`.
44. J. Proença. *Synchronous coordination of distributed components*. PhD thesis, Leiden University, 2011.
45. M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, 1994.
46. The Reo compiler. `https://github.com/ReoLanguage/Reo`.
47. H. Zhu and M. Zhou. Roles in information systems: A survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 38(3):377–396, 2008.

Appendix

In the appendix, we show properties of our novel parallel operator supporting multi-actions and provide detailed explanations on the simple producer-consumer routing example for exogenous PRISM coordination, illustrating application and benefits of our approach.

A Properties of the multi-action parallel operator

Proposition 1. *The parallel composition operator supporting multi-actions is commutative and associative, i.e., for all modules M_1, M_2, and M_3 we have*

(commutativity) $M_1 \parallel M_2 \;\cong\; M_2 \parallel M_1$
(associativity) $\big(M_1 \parallel M_2\big) \parallel M_3 \;\cong\; M_1 \parallel \big(M_2 \parallel M_3\big)$

Here, $\cong$ relates modules whose state-based semantics are isomorphic.

Proof. Commutativity is clear by rules (1), (3), (5a), and (5b): The union of action names ($\cup$), the conjunction of guards ($\wedge$), and the product measure on updates ($\cdot$) are all commutative and the arising command is only open iff both component commands are open, which corresponds to an $\vee$-operation, commutative as well. Furthermore, the conditions in the premises of (1) and (3) are symmetric and (5a) is symmetric to (5b).

Now let us show associativity, i.e., for components M_1, M_2, and M_3 and action sets

$$\begin{aligned} Act_{12} &= Act_1 \cap Act_2 & \qquad Act_{(12)3} &= (Act_1 \cup Act_2) \cap Act_3 \\ Act_{23} &= Act_2 \cap Act_3 & \qquad Act_{1(23)} &= Act_1 \cap (Act_2 \cup Act_3) \end{aligned}$$

we show that

$$\big(M_1|Act_{12}|M_2\big)|Act_{(12)3}|M_3 \;\cong\; M_1|Act_{1(23)}|\big(M_2|Act_{23}|M_3\big) \;.$$

We only have to show one direction since the other direction follows from exchanging the roles of M_1 and M_3 and exploiting commutativity shown above. To this end, we have to show that for every command c in $\big(M_1|Act_{12}|M_2\big)|Act_{(12)3}|M_3$ there is a corresponding command c' of $M_1|Act_{1(23)}|\big(M_2|Act_{23}|M_3\big)$ that has the same impact on the semantics as c. For "having the same impact on the state-based semantics" we also use the notation $c \cong c'$. Here, we use the similar notation as we employed for synchronization sets, e.g., C_{23} is the set of commands in $M_2|Act_{23}|M_3$. Then, let us consider a command of the form $c = \texttt{I}\alpha\texttt{I}\colon g \to u \in C_{(12)3}$. Here, $\texttt{I}\alpha\texttt{I}$ may stand for either $\texttt{[}\alpha\texttt{]}$ or $\texttt{]}\alpha\texttt{[}$. Command c can be the result of one of the rules (1)-(5), which we consider separately by case distinction:

(1) Assume there are commands $c_{12} = \texttt{[}\alpha_{12}\texttt{]}\colon g_{12} \to u_{12} \in C_{12}$ and $c_3 = \texttt{[}\alpha_3\texttt{]}\colon g_3 \to u_3 \in C_3$ such that $\alpha = \alpha_{12} = \alpha_3$, $g = g_{12} \wedge g_3$, and $u = u_{12} \cdot u_3$ with $\alpha_{12} \cap Act_{(12)3} \neq \varnothing$. That is, c can arise from c_{12} and c_3 by rule (1).

(1.1) Assume there are commands $c_1 = [\alpha_1] : g_1 \to u_1 \in C_1$ and $c_2 = [\alpha_2] : g_2 \to u_2 \in C_2$ such that $\alpha_{12} = \alpha_1 = \alpha_2$, $g_{12} = g_1 \wedge g_2$, and $u_{12} = u_1 \cdot u_2$ with $\alpha_1 \cap Act_{12} \neq \varnothing$. That is, c_{12} can arise from c_1 and c_2 by rule (1). Then, $\alpha = \alpha_1 = \alpha_2 = \alpha_3$ and $\varnothing \neq \alpha \subseteq Act_1 \cap Act_2 \cap Act_3$. Applying (1) to commands c_2 and c_3 yields $c_{23} = [\alpha] : g_2 \wedge g_3 \to u_2 \cdot u_3 \in C_{23}$ and applying (1) to commands c_1 and c_{23} yields $c_{1(23)} = [\alpha] : g_1 \wedge (g_2 \wedge g_3) \to u_1 \cdot (u_2 \cdot u_3) \in C_{1(23)}$. Exploiting associativity of $\wedge$ and $\cdot$, we obtain $c \cong c_{1(23)} \in C_{1(23)}$.

(1.2a) Assume there is a command $c_1 = [\alpha_1] : g_1 \to u_1 \in C_1$ such that $g_{12} = g_1$, and $u_{12} = u_1$ with $\alpha_1 \cap Act_{12} = \varnothing$. That is, c_{12} can arise from c_1 by rule (2a). Then, $\alpha = \alpha_1 = \alpha_3$ and $\varnothing \neq \alpha \subseteq (Act_1 \cap Act_3) \setminus Act_2$. As $\alpha_3 \cap Act_{23} = \varnothing$, we can apply (2b) to command c_3, which yields $c_{23} = [\alpha_3] : g_3 \to u_3 \in C_{23}$. Applying (1) to commands c_1 and c_{23} yields $c \cong [\alpha] : g_1 \wedge g_3 \to u_1 \cdot u_3 \in C_{1(23)}$.

(1.2b) Assume there is a command $c_2 = [\alpha_2] : g_2 \to u_2 \in C_2$ such that $g_{12} = g_2$, and $u_{12} = u_2$ with $\alpha_2 \cap Act_{12} = \varnothing$. That is, c_{12} can arise from c_2 by rule (2b). Then, $\alpha = \alpha_2 = \alpha_3$ and $\varnothing \neq \alpha \subseteq (Act_2 \cap Act_3) \setminus Act_1$. The application of (1) to command c_2 and c_3 yields $c_{23} = [\alpha] : g_2 \wedge g_3 \to u_2 \wedge u_3 \in C_{23}$. As $\alpha \cap Act_{12} = \varnothing$, we can apply rule (2b) to command c_{23} and obtain $c \cong [\alpha] : g_2 \wedge g_3 \to u_2 \cdot u_3 \in C_{1(23)}$.

(1.5a) Assume there are commands $c_1 =]\alpha_1[: g_1 \to u_1 \in C_1$ and $c_2 = [\alpha_2] : g_2 \to u_2 \in C_2$ such that $\alpha_{12} = \alpha_1 \cup \alpha_2 \neq \varnothing$ with $\alpha_1 = \alpha_2 \cap Act_1$, $g_{12} = g_1 \wedge g_2$, and $u_{12} = u_1 \cdot u_2$. That is, c_{12} can arise from c_1 and c_2 by rule (5a). Then, $\alpha_{12} = \alpha_2$ and $\alpha = \alpha_2 = \alpha_3$. Furthermore, $\varnothing \neq \alpha \subseteq Act_2 \cap Act_3$. Applying (1) to commands c_2 and c_3 yields $c_{23} = [\alpha] : g_2 \wedge g_3 \to u_2 \cdot u_3 \in C_{23}$. Rule (5a) can be applied to commands c_1 and c_{23} as $\alpha_1 = \alpha_2 \cap Act_1 = \alpha \cap Act_1 \cap (Act_2 \cup Act_3)$ and $\alpha_2 = \alpha_3 \neq \varnothing$. We then obtain $c_{1(23)} = [\alpha] : g_1 \wedge (g_2 \wedge g_3) \to u_1 \cdot (u_2 \cdot u_3) \in C_{1(23)}$. Exploiting associativity of $\wedge$ and $\cdot$, we get $c \cong c_{1(23)} \in C_{1(23)}$.

(1.5b) Assume there are commands $c_1 = [\alpha_1] : g_1 \to u_1 \in C_1$ and $c_2 =]\alpha_2[: g_2 \to u_2 \in C_2$ such that $\alpha_{12} = \alpha_1 \cup \alpha_2 \neq \varnothing$ with $\alpha_2 = \alpha_1 \cap Act_2$, $g_{12} = g_1 \wedge g_2$, and $u_{12} = u_1 \cdot u_2$. That is, c_{12} can arise from c_1 and c_2 by rule (5b). Then, $\alpha_{12} = \alpha_1$ and $\alpha = \alpha_1 = \alpha_3$. Furthermore, $\varnothing \neq \alpha \subseteq Act_1 \cap Act_3$. Rule (5a) can be applied to commands c_2 and c_3 as $\alpha_2 = \alpha_1 \cap Act_2 = \alpha_3 \cap Act_2$ and $\alpha_3 \neq \varnothing$, which yields $c_{23} = [\alpha] : g_2 \wedge g_3 \to u_2 \cdot u_3 \in C_{23}$. Application of rule (1) yields $c_{1(23)} = [\alpha] : g_1 \wedge (g_2 \wedge g_3) \to u_1 \cdot (u_2 \cdot u_3) \in C_{1(23)}$. Exploiting associativity of $\wedge$ and $\cdot$, we get $c \cong c_{1(23)} \in C_{1(23)}$.

(2a) Assume there is a command $c_{12} = [\alpha_{12}] : g_{12} \to u_{12} \in C_{12}$ such that $\alpha = \alpha_{12}$ with $\alpha_{12} \cap Act_{(12)3} = \alpha_{12} \cap Act_3 = \varnothing$, $g = g_{12}$, and $u = u_{12}$. That is, c can arise from c_{12} by rule (2a).

(2a.1) Assume there are commands $c_1 = [\alpha_1] : g_1 \to u_1 \in C_1$ and $c_2 = [\alpha_2] : g_2 \to u_2 \in C_2$ such that $\alpha_{12} = \alpha_1 = \alpha_2$, $g_{12} = g_1 \wedge g_2$, and $u_{12} = u_1 \cdot u_2$ with $\alpha_1 \cap Act_2 \neq \varnothing$. That is, c_{12} can arise from c_1 and c_2 by rule (1). Then, $\alpha = \alpha_1 = \alpha_2 = \alpha_{12}$ and $\varnothing \neq \alpha \subseteq (Act_1 \cap Act_2) \setminus Act_3$. As $\alpha_2 \cap Act_3 = \varnothing$, rule (2a) can be applied to command c_2, yielding

$c_{23} = [\alpha] : g_2 \to u_2 \in C_{23}$. Applying (1) to commands c_1 and c_{23} yields $c \cong [\alpha] : g_1 \wedge g_2 \to u_1 \cdot u_2 \in C_{1(23)}$.

(2a.2a) Assume there is a command $c_1 = [\alpha_1] : g_1 \to u_1 \in C_1$ such that $\alpha_{12} = \alpha_1$, $g_{12} = g_1$, and $u_{12} = u_1$ with $\alpha_1 \cap Act_2 = \varnothing$. That is, c_{12} can arise from c_1 by rule (2a). Hence, $\alpha_1 \cap (Act_2 \cup Act_3) = \varnothing$ and we can apply rule (2a) to c_1 and obtain $c \cong c_1 \in C_{1(23)}$.

(2a.2b) Assume there is a command $c_2 = [\alpha_2] : g_2 \to u_2 \in C_2$ such that $\alpha_{12} = \alpha_2$, $g_{12} = g_2$, and $u_{12} = u_2$ with $\alpha_2 \cap Act_1 = \varnothing$. That is, c_{12} can arise from c_2 by rule (2b). Hence, $\alpha_2 \cap Act_3 = \varnothing$, so that rule (2a) can be applied to c_2, yielding $c_2 \in C_{23}$. Application of rule (2b) onto c_2 is possible due to $\alpha_2 \cap Act_1 = \varnothing$, which yields $c \cong c_2 \in C_{1(23)}$.

(2a.5a) Assume there are commands $c_1 =]\alpha_1[: g_1 \to u_1 \in C_1$ and $c_2 = [\alpha_2] : g_2 \to u_2 \in C_2$ such that $\alpha_{12} = \alpha_1 \cup \alpha_2 \neq \varnothing$ with $\alpha_1 = \alpha_2 \cap Act_1$, $g_{12} = g_1 \wedge g_2$, and $u_{12} = u_1 \cdot u_2$. That is, c_{12} can arise from c_1 and c_2 by rule (5a). Then, $\alpha_{12} = \alpha_2 = \alpha$. Furthermore, $\varnothing \neq \alpha \subseteq Act_2 \setminus Act_3$. As $\alpha_2 \cap Act_3 = \varnothing$, rule (2a) can be applied to command c_2, yielding $c_{23} = [\alpha] : g_2 \to u_2 \in C_{23}$. Applying (5a) to commands c_1 and c_{23} yields $c \cong [\alpha] : g_1 \wedge g_2 \to u_1 \cdot u_2 \in C_{1(23)}$.

(2a.5b) Assume there are commands $c_1 = [\alpha_1] : g_1 \to u_1 \in C_1$ and $c_2 =]\alpha_2[: g_2 \to u_2 \in C_2$ such that $\alpha_{12} = \alpha_1 \cup \alpha_2 \neq \varnothing$ with $\alpha_2 = \alpha_1 \cap Act_2$, $g_{12} = g_1 \wedge g_2$, and $u_{12} = u_1 \cdot u_2$. That is, c_{12} can arise from c_1 and c_2 by rule (5b). Then, $\alpha_{12} = \alpha_1 = \alpha$. Furthermore, $\varnothing \neq \alpha \subseteq Act_1 \setminus Act_3$. Rule (4a) can be applied to the command c_2 as $\alpha_1 \cap Act_3 = \varnothing$ and thus $\alpha_1 \cap Act_2 \cap Act_3 = \alpha_2 \cap Act_3 = \varnothing$, which yields $c_{23} =]\alpha[: g_2 \to u_2 \in C_{23}$. Application of rule (5b) yields $c \cong [\alpha] : g_1 \wedge g_2 \to u_1 \cdot u_2 \in C_{1(23)}$.

(2b) Assume there is a command $c_3 = [\alpha_3] : g_3 \to u_3 \in C_3$ such that $\alpha_3 \cap Act_{(12)3} = \alpha_3 \cap (Act_1 \cup Act_2) = \varnothing$ and $\alpha = \alpha_3$, $g = g_3$, and $u = u_3$. That is, c can arise from c_3 by rule (2b). Then, due to $\alpha_3 \cap Act_2 = \varnothing$, rule (2b) yields $c_3 \in C_{23}$. By the same argument, as $\alpha_3 \cap Act_1 = \varnothing$, rule (2b) finally yields $c \cong c_3 \in C_{1(23)}$.

(3) Assume there are commands $c_{12} =]\alpha_{12}[: g_{12} \to u_{12} \in C_{12}$ and $c_3 =]\alpha_3[: g_3 \to u_3 \in C_3$ such that $\alpha = \alpha_{12} \cup \alpha_3$, $g = g_{12} \wedge g_3$, and $u = u_{12} \cdot u_3$ with $\alpha_{12} \cap Act_{(12)3} = \alpha_3 \cap Act_{(12)3}$. That is, c can arise from c_{12} and c_3 by rule (3). Hence, $\alpha_{12} \cap Act_3 = \alpha_3 \cap (Act_1 \cup Act_2)$.

(3.3) Assume commands $c_1 =]\alpha_1[: g_1 \to u_1 \in C_1$ and $c_2 =]\alpha_2[: g_2 \to u_2 \in C_2$ such that $\alpha_1 \cap Act_2 = \alpha_2 \cap Act_1$, $\alpha_{12} = \alpha_1 \cup \alpha_2$, $g_{12} = g_1 \wedge g_2$, and $u_{12} = u_1 \cdot u_2$. That is, c_{12} can arise from c_1 and c_2 by rule (3). We show that then, $\alpha_2 \cap Act_3 = \alpha_3 \cap Act_2$:

($\subseteq$): Let $a \in \alpha_2 \cap Act_3$, then $a \in \alpha_3$ due to $(\alpha_1 \cup \alpha_2) \cap Act_3 = \alpha_3 \cap (Act_1 \cup Act_2)$. Thus, $a \in \alpha_3 \cap Act_2$.

($\supseteq$): Let $a \in \alpha_3 \cap Act_2$, then $a \in \alpha_1 \cup \alpha_2$ due to $(\alpha_1 \cup \alpha_2) \cap Act_3 = \alpha_3 \cap (Act_1 \cup Act_2)$. As $a \in Act_2$ and $\alpha_1 \cap Act_2 = \alpha_2 \cap Act_1$, $a \in \alpha_1$ implies $a \in \alpha_2$ and thus, $a \in \alpha_2$ in any case. Hence, $a \in \alpha_2 \cap Act_3$.

Thus, we can apply rule (3) to c_2 and c_3, yielding $c_{23} =]\alpha_2 \cup \alpha_3[: g_2 \wedge g_3 \to u_2 \cdot u_3 \in C_{23}$. We show that $\alpha_1 \cap (Act_2 \cup Act_3) = (\alpha_2 \cup \alpha_3) \cap Act_1$:

($\subseteq$): Let $a \in \alpha_1 \cap (Act_2 \cup Act_3)$. In case $a \in Act_2$, then due to $\alpha_1 \cap Act_2 = \alpha_2 \cap Act_1$ we have that $a \in \alpha_2$ and hence $a \in (\alpha_2 \cup \alpha_3) \cap Act_1$. Otherwise, $a \in Act_3$ and due to $(\alpha_1 \cup \alpha_2) \cap Act_3 = \alpha_3 \cap (Act_1 \cup Act_2)$ we obtain $a \in \alpha_3$, also resulting in $a \in (\alpha_2 \cup \alpha_3) \cap Act_1$.

($\supseteq$): Let $a \in (\alpha_2 \cup \alpha_3) \cap Act_1$. If $a \in \alpha_2$, then due to $\alpha_1 \cap Act_2 = \alpha_2 \cap Act_1$ we have that $a \in \alpha_1$ and thus, $a \in \alpha_1 \cap (Act_2 \cup Act_3)$. Otherwise, if $a \in \alpha_3$ and $a \notin \alpha_2$ we have that $a \in \alpha_1$ due to $(\alpha_1 \cup \alpha_2) \cap Act_3 = \alpha_3 \cap (Act_1 \cup Act_2)$. Hence, $a \in \alpha_1 \cap (Act_2 \cup Act_3)$.

Thus, we can again apply rule (3) to c_1 and c_{23} yields $c_{1(23)} =]\alpha_1 \cup (\alpha_2 \cup \alpha_3)[: g_1 \wedge (g_2 \wedge g_3) \to u_1 \cdot (u_2 \cdot u_3) \in C_{1(23)}$. Exploiting associativity of $\cup$, $\wedge$, and $\cdot$, we get $c \cong c_{1(23)} \in C_{1(23)}$.

(3.4a) Assume that there is a command $c_1 =]\alpha_1[: g_1 \to u_1 \in C_1$ such that $\alpha_1 \cap Act_2 = \varnothing$, $\alpha_{12} = \alpha_1$, $g_{12} = g_1$, and $u_{12} = u_1$. That is, c_{12} can arise from c_1 by rule (4a). Then $\alpha_1 \cap Act_3 = \alpha_3 \cap Act_1$ and as $\alpha_1 \cap Act_2 =$ we have $\alpha_3 \cap Act_2 = \varnothing$. Applying rule (4b) on c_3 yields $c_3 \in C_{23}$. $\alpha_1 \cap Act_3 = \alpha_3 \cap Act_1$ yields applicability of rule (3) onto c_1 and c_3 towards $c_{13} =]\alpha_1 \cup \alpha_3[: g_1 \wedge g_3 \to u_1 \cdot u_3 \in C_{1(23)}$ and hence $c \cong c_{13} \in C_{1(23)}$.

(3.4b) Assume that there is a command $c_2 =]\alpha_2[: g_2 \to u_2 \in C_2$ such that $\alpha_2 \cap Act_1 = \varnothing$, $\alpha_{12} = \alpha_2$, $g_{12} = g_2$, and $u_{12} = u_2$. That is, c_{12} can arise from c_2 by rule (4b). Since $\alpha_2 \cap Act_1 = \varnothing$ we get $\alpha_2 \cap Act_3 = \alpha_3 \cap Act_2$ and rule (3) onto c_2 and c_3 yields $c_{23} =]\alpha_2 \cup \alpha_3[: g_2 \wedge g_3 \to u_2 \cdot u_3 \in C_{23}$. Furthermore, $\alpha_3 \cap Act_1 = \varnothing$ (otherwise $\alpha_2 \cap Act_1 \neq \varnothing$) and thus, as $(\alpha_2 \cup \alpha_3) \cap Act_1 = \varnothing$, rule (4b) yields $c \cong c_{23} \in C_{1(23)}$.

(4a) Assume there is a command $c_{12} =]\alpha_{12}[: g_{12} \to u_{12} \in C_{12}$ with $\alpha_{12} \cap Act_3 = \varnothing$ such that $\alpha = \alpha_{12}$, $g = g_{12}$, and $u = u_{12}$. That is, c can arise from c_{12} by rule (4a).

(4a.3) Assume that c_{12} can arise from commands $c_1 =]\alpha_1[: g_1 \to u_1 \in C_1$ and $c_2 =]\alpha_2[: g_2 \to u_2 \in C_2$ by rule (3), i.e., $\alpha_1 \cap Act_2 = \alpha_2 \cap Act_1$, $\alpha_{12} = \alpha_1 \cup \alpha_2$, $g_{12} = g_1 \wedge g_2$, and $u_{12} = u_1 \cdot u_2$. Then, as $(\alpha_1 \cup \alpha_2) \cap Act_3 = \varnothing$, we get $\alpha_2 \cap Act_3 = \varnothing$ and thus, rule (4a) provides $c_2 \in C_{23}$. Rule (3) applied on c_1 and c_2 yields $c \in C_{1(23)}$.

(4a.4a) Assume that c_{12} can arise from $c_1 =]\alpha_1[: g_1 \to u_1 \in C_1$ by rule (4a), i.e., $\alpha_1 \cap Act_2 = \varnothing$, $\alpha_{12} = \alpha_1$, $g_{12} = g_1$, and $u_{12} = u_1$. As $\alpha_1 \cap Act_3 = \varnothing$ and $\alpha_1 \cap Act_2 = \varnothing$, we have $\alpha_3 \cap (Act_2 \cup Act_3) = \varnothing$ and rule (4a) for the composition of M_1 with $M_2 \parallel M_3$ is applicable, leading to $c_1 = c \in C_{1(23)}$.

(4a.4b) Assume that c_{12} can arise from $c_2 =]\alpha_2[: g_2 \to u_2 \in C_2$ by rule (4b), i.e., $\alpha_2 \cap Act_1 = \varnothing$, $\alpha_{12} = \alpha_2$, $g_{12} = g_2$, and $u_{12} = u_2$. As $\alpha_2 \cap Act_3 = \varnothing$ we can apply rule (4a) and get $c_2 \in C_{23}$. Then, due to $\alpha_2 \cap Act_1 = \varnothing$, we can apply rule (4b) to obtain $c_2 = c \in C_{1(23)}$.

(4b) Assume there is a command $c_3 =]\alpha_3[: g_3 \to u_3 \in C_3$ with $\alpha_3 \cap Act_{(12)3} = \alpha_3 \cap (Act_1 \cup Act_2) = \varnothing$ such that $\alpha = \alpha_3$, $g = g_3$, and $u = u_3$. That is, c can arise from c_3 by applying rule (4b). Then, $\alpha_3 \cap Act_2 = \varnothing$ and applying rule (4b) twice yields $c_3 = c \in C_{1(23)}$.

(5a) Assume there are commands $c_{12} =]\alpha_{12}[: g_{12} \to u_{12} \in C_{12}$ and $c_3 = [\alpha_3] : g_3 \to u_3 \in C_3$ such that $\alpha = \alpha_3 \neq \varnothing$, $g = g_{12} \wedge g_3$, and $u = u_{12} \cdot u_3$ with $\alpha_{12} = \alpha_3 \cap Act_{(12)3}$. That is, rule c can arise from c_{12} and c_3 by applying rule (5a).

(5a.3) Assume commands $c_1 =]\alpha_1[: g_1 \to u_1 \in C_1$ and $c_2 =]\alpha_2[: g_2 \to u_2 \in C_2$ such that $\alpha_1 \cap Act_2 = \alpha_2 \cap Act_1$, $\alpha_{12} = \alpha_1 \cup \alpha_2$, $g_{12} = g_1 \wedge g_2$, and $u_{12} = u_1 \cdot u_2$. That is, c_{12} can arise from c_1 and c_2 by rule (3). We first show that $\alpha_2 = \alpha_3 \cap Act_2$:

($\subseteq$): Let $a \in \alpha_2$. Then, $a \in \alpha_3 \cap \alpha_2$ clearly by $\alpha_1 \cup \alpha_2 = \alpha_3 \cap (Act_1 \cup Act_2)$.

($\supseteq$): Let $a \in \alpha_3 \cap Act_2$. As $\alpha_1 \cup \alpha_2 = \alpha_3 \cap (Act_1 \cup Act_2)$, either $a \in \alpha_1$ or $a \in \alpha_2$. In case $a \in \alpha_2$, we are done. Otherwise, $a \in \alpha_1$ and we also have $a \in \alpha_2$ as $a \in Act_1 \cap Act_2$ and $\alpha_1 \cap Act_2 = \alpha_2 \cap Act_1$.

Hence, as also $\alpha_3 \neq \varnothing$, rule (5a) applied on c_2 and c_3 yields $c_{23} = [\alpha_2 \cup \alpha_3] : g_2 \wedge g_3 \to u_2 \cdot u_3 \in C_{23}$.

Now, we show that $\alpha_1 = (\alpha_2 \cup \alpha_3) \cap Act_1$:

($\subseteq$): Let $a \in \alpha_1$. In case $a \in Act_2$, then due to $\alpha_1 \cap Act_2 = \alpha_2 \cap Act_1$ we have that $a \in \alpha_2$ and hence $a \in (\alpha_2 \cup \alpha_3) \cap Act_1$. Otherwise, $a \in Act_3 \setminus Act_2$ and due to $\alpha_1 \cup \alpha_2 = \alpha_3 \cap (Act_1 \cup Act_2)$ we obtain $a \in \alpha_3$, also resulting in $a \in (\alpha_2 \cup \alpha_3) \cap Act_1$.

($\supseteq$): Let $a \in (\alpha_2 \cup \alpha_3) \cap Act_1$. If $a \in \alpha_2$, then due to $\alpha_1 \cap Act_2 = \alpha_2 \cap Act_1$ we have that $a \in \alpha_1$. Otherwise, if $a \in \alpha_3$ and $a \notin \alpha_2$ we have that $a \in \alpha_1$ due to $\alpha_1 \cup \alpha_2 = \alpha_3 \cap (Act_1 \cup Act_2)$. Hence, $a \in \alpha_1 \cap (Act_2 \cup Act_3)$.

Thus, we can apply rule (5a) to c_1 and c_{23} to obtain $c_{1(23)} = [\alpha_1 \cup (\alpha_2 \cup \alpha_3)] : g_1 \wedge (g_2 \wedge g_3) \to u_1 \cdot (u_2 \cdot u_3) \in C_{1(23)}$. Exploiting associativity of $\cup$, $\wedge$, and $\cdot$, we get $c \cong c_{1(23)} \in C_{1(23)}$.

(5a.4a) Assume that c_{12} can arise from $c_1 =]\alpha_1[: g_1 \to u_1 \in C_1$ by rule (4a), i.e., $\alpha_1 \cap Act_2 = \varnothing$, $\alpha_{12} = \alpha_1$, $g_{12} = g_1$, and $u_{12} = u_1$. Then, $\alpha_3 \cap Act_2 = \varnothing$ as otherwise there is an $a \in \alpha_3 \cap Act_2$ and by $\alpha_1 = \alpha_3 \cap (Act_1 \cup Act_2)$ we have $a \in \alpha_1$, contradicting $\alpha_1 \cap Act_2 = \varnothing$. Thus, rule (2b) is applicable to c_3, such that $c_3 \in C_{23}$. By $\alpha_1 = \alpha_3 \cap (Act_1 \cup Act_2)$ and $\alpha_1 \cap Act_2 = \varnothing$ we obtain $\alpha_1 = \alpha_3 \cap Act_1$. Thus, we can apply rule (5a) onto c_1 and c_3 towards $c_{13} = [\alpha_1 \cup \alpha_3] : g_1 \wedge g_3 \to u_1 \cdot u_3 \in C_{1(23)}$ and obtain $c \cong c_{13} \in C_{1(23)}$.

(5a.4b) Assume that c_{12} can arise from $c_2 =]\alpha_2[: g_2 \to u_2 \in C_2$ by rule (4b), i.e., $\alpha_2 \cap Act_1 = \varnothing$, $\alpha_{12} = \alpha_2$, $g_{12} = g_2$, and $u_{12} = u_2$. As $\alpha_2 = \alpha_3 \cap (Act_1 \cup Act_2)$ and $\alpha_2 \cap Act_1 = \varnothing$, we have $\alpha_2 = \alpha_3 \cap Act_2$. Since furthermore $\alpha_3 \neq \varnothing$, we can apply rule (5a) onto c_2 and c_3 towards $c_{23} = [\alpha_2 \cup \alpha_3] : g_2 \wedge g_3 \to u_2 \cdot u_3 \in C_{23}$. $(\alpha_2 \cup \alpha_3) \cap Act_1 = \alpha_3 \cap Act_1$ as $\alpha_2 \cap Act_1 = \varnothing$. Furthermore, $\alpha_3 \cap Act_1 = \varnothing$ as otherwise by there would be an $a \in \alpha_3 \cap Act_1$ for which also $a \in \alpha_2 = \alpha_3 \cap (Act_1 \cup Act_2)$, contradicting $\alpha_2 \cap Act_1 = \varnothing$. Hence, we can apply rule (2b) onto c_{23} and obtain $c \cong c_{23} \in C_{1(23)}$.

(5b) Assume there are commands $c_{12} = [\alpha_{12}] : g_{12} \to u_{12} \in C_{12}$ and $c_3 =]\alpha_3[: g_3 \to u_3 \in C_3$ such that $\alpha = \alpha_{12} \neq \varnothing$, $g = g_{12} \wedge g_3$, and $u = u_{12} \cdot u_3$ with $\alpha_3 = \alpha_{12} \cap Act_{(12)3}$. That is, rule c can arise from c_{12} and c_3 by applying rule (5b).

(5b.1) Assume there are commands $c_1 = [\alpha_1] : g_1 \to u_1 \in C_1$ and $c_2 = [\alpha_2] : g_2 \to u_2 \in C_2$ such that $\alpha_{12} = \alpha_1 = \alpha_2 \neq \varnothing$, $g_{12} = g_1 \wedge g_2$, and $u_{12} = u_1 \cdot u_2$ with $\alpha_1 \cap Act_2 \neq \varnothing$. That is, c_{12} can arise from c_1 and c_2 by rule (1). Then, $\alpha = \alpha_1 = \alpha_2 = \alpha_{12}$ and $\alpha_3 = \alpha_2 \cap Act_3$ and we can apply rule (5b) to obtain $c_{23} = [\alpha_2 \cup \alpha_3] : g_2 \wedge g_3 \to u_2 \cdot u_3 \in C_{23}$. Since $\alpha_2 \cap Act_3 = \alpha_3$ we have $\alpha_3 \subseteq \alpha_2 = \alpha_1$ and hence, $\alpha_1 = \alpha_2 \cup \alpha_3$. In combination with $\alpha_1 \cap Act_2 \neq \varnothing$ we can apply rule (1) onto c_1 and c_{23} to obtain $c_{1(23)} = [\alpha_1 \cup (\alpha_2 \cup \alpha_3)] : g_1 \wedge (g_2 \wedge g_3) \to u_1 \cdot (u_2 \cdot u_3) \in C_{1(23)}$. Exploiting associativity of $\cup$, $\wedge$, and $\cdot$, we get $c \cong c_{1(23)} \in C_{1(23)}$.

(5b.2a) Assume there is a command $c_1 = [\alpha_1] : g_1 \to u_1 \in C_1$ such that $\alpha_{12} = \alpha_1$, $g_{12} = g_1$, and $u_{12} = u_1$ with $\alpha_1 \cap Act_2 = \varnothing$. That is, c_{12} can arise from c_1 by rule (2a). Hence, $\alpha_3 = \alpha_1 \cap Act_3$ and thus, $\alpha_3 \subseteq \alpha_1$, which leads to $\alpha_3 \cap Act_2 = \varnothing$ as $\alpha_1 \cap Act_2 = \varnothing$. Application of rule (4b) onto c_3 yields $c_3 \in C_{23}$. As $\alpha_1 \neq \varnothing$ and $\alpha_3 = \alpha_1 \cap Act_3$, we can apply rule (5b) onto c_1 and c_3 towards $c_{13} = [\alpha_1 \cup \alpha_3] : g_1 \wedge g_3 \to u_1 \cdot u_3 \in C_{1(23)}$ and obtain $c \cong c_{13} \in C_{1(23)}$.

(5b.2b) Assume there is a command $c_2 = [\alpha_2] : g_2 \to u_2 \in C_2$ such that $\alpha_{12} = \alpha_2$, $g_{12} = g_2$, and $u_{12} = u_2$ with $\alpha_2 \cap Act_1 = \varnothing$. That is, c_{12} can arise from c_2 by rule (2b). As $\alpha_2 \cap Act_1 = \varnothing$, $\alpha_3 = \alpha_2 \cap Act_3$ and hence, rule (5b) can be applied to c_2 and c_3 towards $c_{23} = [\alpha_2 \cup \alpha_3] : g_2 \wedge g_3 \to u_2 \cdot u_3 \in C_{23}$. Then, $(\alpha_2 \cup \alpha_3) \cap Act_1 = (\alpha_2 \cup (\alpha_2 \cap Act_3)) \cap Act_1 = \alpha_2 \cap Act_1 = \varnothing$. Thus, rule (2b) is applicable to c_{23} and yields $c \cong c_{23} \in C_{1(23)}$.

□

B Simple routing example for exogenous Prism coordination

We provide here the detailed models for the producer-consumer example from Sections 3 and 4, to illustrate the different modeling approaches.

B.1 Standard Prism, endogenous synchronization

Consider Fig. 5, which depicts a system with three producers and three consumers. Whenever one of the producers (or consumers) is active, there is a 10% chance that the producer (resp. consumer) crashes and will not be available afterwards.

Our goal is to ensure that exactly one producer and one consumer are active at the same time. To model this, we have to hard-code the synchronization into the producer and consumer modules, via one action for each producer/consumer interaction. In particular, we have to duplicate the guarded commands in the producers and consumers to account for all possible synchronization variants, providing unique action names for each possible combination. All these action names have to be renamed for the instantiation of the second and third producer/consumer. Clearly, modifying the coordination between the producers

```
module Producer1
  p1_broken : bool init false;

  [p1_c1] !p1_broken -> 0.1:(p1_broken'=true) + 0.9:(p1_broken'=false);
  [p1_c2] !p1_broken -> 0.1:(p1_broken'=true) + 0.9:(p1_broken'=false);
  [p1_c3] !p1_broken -> 0.1:(p1_broken'=true) + 0.9:(p1_broken'=false);
endmodule

module Consumer1
  c1_broken : bool init false;

  [p1_c1] !c1_broken -> 0.1:(c1_broken'=true) + 0.9:(c1_broken'=false);
  [p2_c1] !c1_broken -> 0.1:(c1_broken'=true) + 0.9:(c1_broken'=false);
  [p3_c1] !c1_broken -> 0.1:(c1_broken'=true) + 0.9:(c1_broken'=false);
endmodule

module Producer2
  = Producer1 [p1_broken = p2_broken, p1_c1=p2_c1, p1_c2=p2_c2, p1_c3=p2_c3]
endmodule

module Producer3
  = Producer1 [p1_broken = p3_broken, p1_c1=p3_c1, p1_c2=p3_c2, p1_c3=p3_c3]
endmodule

module Consumer2
  = Consumer1 [c1_broken = c2_broken, p1_c1=p1_c2, p2_c1=p2_c2, p3_c1=p3_c2]
endmodule

module Consumer3
  = Consumer1 [c1_broken = c3_broken, p1_c1=p1_c3, p2_c1=p2_c3, p3_c1=p3_c3]
endmodule
```

Fig. 5. PRISM language model description for a system with three producers and three consumers and with endogenous coordination/synchronization.

and consumers, e.g., to obtain a variant or add more complex synchronization schemes requires the user to touch all parts of the system manually, which can be tedious and error prone.

B.2 Prism with multi-actions, exogenous coordination

Using our extensions to PRISM, we can model the same system more conveniently using exogenous coordination. Consider the PRISM language description in Fig. 6. There, we define a module template for a producer and for a consumer. Note that they each consist of a single guarded command, with a `p`-action for production and a `c`-action for consumption. During instantiation, we use the `varprefix` rule to disambiguate the state variables and rename the producer's/consumer's action to a unique action name.

The actual synchronization is then achieved using two modules that provide the coordination glue code. The first module, a merger, has a non-deterministic choice between `p1`, `p2` and `p3` and synchronizes these actions with an `n`-action. The second module, for routing, has a non-deterministic choice between the actions `c1`, `c2` and `c3`, synchronizing as well on action `n`. In the parallel composition of all the modules, this results in the same synchronization as in the system above. For example, the multi-action `p1,n,c2` corresponds to the action `p1_c2` in the PRISM model with hardcoded endogenous coordination.

Using this modeling approach, we have decoupled the coordination between the computation parts of the system. It is easy to see that additional producers and consumers can easily be added and alternative coordination schemes can be used by replacing the coordination glue code. Such changes can be made without modifying the definitions of the producers and consumer itself.

B.3 Prism with Reo

Using REO, we can model the coordination between the producer and consumer modules using a network of channels, which can then be compiled into a PRISM language representation. Fig. 8 provides the input to the REOCOMPILER in the format of its textual REO input language of the REO network depicted in Fig. 7.

We first import channels and circuitry from the REO library and then provide a description of the REO network, consisting of the instantiation of the producer and consumer components and synchronous channels to connect the producers' ports to a REO node `n`, which provides the *merge* semantics, i.e., ensuring that exactly one of the producers can be active at the same time. An exclusive router then connects node `n` to the consumers inputs, providing the non-deterministic choice between the consumers. Additionally, we introduce *wrappers* for the PRISM language definitions of the producer and consumer module templates. Here, we define the external interface, i.e., the names of the actions in the module with the additional information whether such a port should be an input port (`a?`) or an output port (`a!`) in the REO network. Then, we provide a reference to the module template name for the PRISM target language. These

```
module Producer as template
  broken : bool init false;

  ]p[ !broken -> 0.1:(broken'=true) + 0.9:(broken'=false);
endmodule

module Consumer as template
  broken : bool init false;

  ]c[ !broken -> 0.1:(broken'=true) + 0.9:(broken'=false);
endmodule

// instantiate producers (rename state and action)
module Producer1 = Producer (varprefix=p1_) [p=p1] endmodule
module Producer2 = Producer (varprefix=p2_) [p=p2] endmodule
module Producer3 = Producer (varprefix=p3_) [p=p3] endmodule

// instantiate consumers (rename state and action)
module Consumer1 = Consumer (varprefix=c1_) [c=c1] endmodule
module Consumer2 = Consumer (varprefix=c2_) [c=c2] endmodule
module Consumer3 = Consumer (varprefix=c3_) [c=c3] endmodule

// sync one of p1, p2, p3 with action n
module merge
  ]p1,n[ true -> true;
  ]p2,n[ true -> true;
  ]p3,n[ true -> true;
endmodule

// sync n with one of c1, c2, c3
module route
  ]n,c1[ true -> true;
  ]n,c2[ true -> true;
  ]n,c3[ true -> true;
endmodule
```

Fig. 6. Extended PRISM language model description for a system with three producers and three consumers, using exogenous coordination via multi-actions.

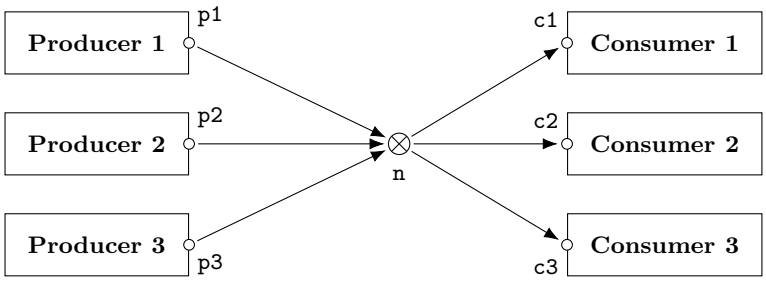

Fig. 7. Reo network, coordinating producers and consumers via an x-router.

```
import reo.xrouter3;
import reo.sync;

producerConsumer(p1,p2,p3,c1,c2,c3) {
   prod1 = producer(p1)
   prod2 = producer(p2)
   prod3 = producer(p3)

   cons1 = consumer(c1)
   cons2 = consumer(c2)
   cons3 = consumer(c3)

   sync(p1, n)
   sync(p2, n)
   sync(p3, n)
   xrouter3(n, c1, c2, c3)
}

producer(a!) {
   #PRISM
   "Producer"
}

consumer(a?) {
   #PRISM
   "Consumer"
}
```

Fig. 8. Textual Reo description of the Reo network in Fig. 7, coordinating the producers and consumers given as Prism modules.

```
module Producer as template
  broken : bool init false;

  ]a[ !broken -> 0.1:(broken'=true) + 0.9:(broken'=false);
endmodule

module Consumer as template
  broken : bool init false;

  ]a[ !broken -> 0.1:(broken'=true) + 0.9:(broken'=false);
endmodule
```

Fig. 9. Extended PRISM language module templates for a producer and a consumer, used as a library during the REOCOMPILER compilation.

definitions are provided by means of a library PRISM file (depicted in Fig. 9), that is linked during the REOCOMPILER compilation.

Figure 10 shows the extended PRISM language model as generated by our extended version of the REOCOMPILER, using the monolithic compilation option. Here, the coordination glue code is internally composed by the REOCOMPILER and output as a single protocol PRISM module. The guards that appear in the commands are due to side effects in the automatic generation of data-abstract glue code. Likewise, the constant `N`, which encodes the range of values for memory cells in FIFO channels, is set to 1 for data-abstract glue code.

In the alternative compilation mode, the REOCOMPILER does not compose the REO network into one monolithical module, but instead translates each part of the REO network (channels and nodes) into individual PRISM modules. Figure 11 shows the generated PRISM language model for this compilation mode. As can be seen, the coordination glue code in the REO network is now represented by multiple modules, leaving the composition to PRISM.

Using the textual description of a REO network and later compilation via the REOCOMPILER allows leveraging the rich coordination patterns available in the REO language. In particular, it becomes easy to replace the coordination with different variants, just by replacing the required parts of the REO network.

```
mdp

// constant for the upper bound on the data domain of connector variables
const N = 1;

// ---- included from producerConsumer-library.prism --------
module Producer as template
  broken : bool init false;

  ]a[ !broken -> 0.1:(broken'=true) + 0.9:(broken'=false);
endmodule

module Consumer as template
  broken : bool init false;

  ]a[ !broken -> 0.1:(broken'=true) + 0.9:(broken'=false);
endmodule
// ---------------------------------------

module prod1 = Producer (varprefix=prod1_) [ a = p1 ] endmodule
module prod2 = Producer (varprefix=prod2_) [ a = p2 ] endmodule
module prod3 = Producer (varprefix=prod3_) [ a = p3 ] endmodule
module cons1 = Consumer (varprefix=cons1_) [ a = c1 ] endmodule
module cons2 = Consumer (varprefix=cons2_) [ a = c2 ] endmodule
module cons3 = Consumer (varprefix=cons3_) [ a = c3 ] endmodule

module Protocol1
    ]p1, c3[ (true & !(1 = 0)) & 1 = 1 -> true;
    ]p1, c2[ (true & !(1 = 0)) & 1 = 1 -> true;
    ]p1, c1[ (true & !(1 = 0)) & 1 = 1 -> true;
    ]p2, c3[ (true & !(1 = 0)) & 1 = 1 -> true;
    ]p2, c2[ (true & !(1 = 0)) & 1 = 1 -> true;
    ]p2, c1[ (true & !(1 = 0)) & 1 = 1 -> true;
    ]p3, c3[ (true & !(1 = 0)) & 1 = 1 -> true;
    ]p3, c2[ (true & !(1 = 0)) & 1 = 1 -> true;
    ]p3, c1[ (true & !(1 = 0)) & 1 = 1 -> true;
endmodule
```

Fig. 10. Extended PRISM language model generated for the producer/consumer REO network (monolithic variant, single protocol module).

```
mdp

// constant for the upper bound on the data domain of connector variables
const N = 1;

// ---- included from producerConsumer-library.prism --------
module Producer as template
  broken : bool init false;
  ]a[ !broken -> 0.1:(broken'=true) + 0.9:(broken'=false);
endmodule

module Consumer as template
  broken : bool init false;
  ]a[ !broken -> 0.1:(broken'=true) + 0.9:(broken'=false);
endmodule
// ---------------------------------------

module prod1 = Producer (varprefix=prod1_) [ a = p1 ] endmodule
module prod2 = Producer (varprefix=prod2_) [ a = p2 ] endmodule
module prod3 = Producer (varprefix=prod3_) [ a = p3 ] endmodule
module cons1 = Consumer (varprefix=cons1_) [ a = c1 ] endmodule
module cons2 = Consumer (varprefix=cons2_) [ a = c2 ] endmodule
module cons3 = Consumer (varprefix=cons3_) [ a = c3 ] endmodule

module sync1
    ]p1, _1_0[ !(1 = 0) & 1 = 1 -> true;
endmodule

module sync2
    ]p2, _1_1[ !(1 = 0) & 1 = 1 -> true;
endmodule

module sync3
    ]p3, _1_2[ !(1 = 0) & 1 = 1 -> true;
endmodule

module xrouter31
    ]_1, c1[ !(1 = 0) & 1 = 1 -> true;
    ]_1, c2[ !(1 = 0) & 1 = 1 -> true;
    ]_1, c3[ !(1 = 0) & 1 = 1 -> true;
endmodule

module node1
    ]_1_0, _1[ true & 1 = 1 -> true;
    ]_1_1, _1[ true & 1 = 1 -> true;
    ]_1_2, _1[ true & 1 = 1 -> true;
endmodule
```

Fig. 11. Extended PRISM language model generated for the producer/consumer REO network (compositional variant, individual modules for the different parts of the coordination glue code).

B.4 Reward monitors in Reo and Prism

To illustrate the use of reward monitors, we use the textual description of a Reo network shown in Fig. 12. The definitions for the producer and consumer module templates are is as in Fig. 9. Here, we instantiate one producer and one consumer and connect them with two FIFO channels (with capacity 1), chained one after the other. We are interested in measuring the activity at various locations of the network using reward monitors. For this, we provide definitions for two reward monitor components. The first one, `localActivity` assigns a reward of 1 whenever it's input port is active. We instantiate one of these monitors, attach it to the `n` node between the FIFO channels and assign it the name `nodeMonitor`. The second monitor, `activity` provides global reward accumulation under the name `"activity"`. Thus, the two instantiations, connected to the output port of the producer and the input port of the consumer are counted in the same reward structure.

Fig. 13 then shows the generated Prism language model description, generated in the compositional mode. As can be seen, the corresponding reward structure `"nodeMonitor"` monitors the activity at the action name assigned for the Reo node `n`, and the two reward monitors attached to `p` and `c` result in a combined reward structure that represents the addition of the two reward rules.

```
import reo.fifo1;

fifoRewards(p,c,n) {
   prod = producer(p)
   cons = consumer(c)

   f1 = fifo1(p,n)
   f2 = fifo1(n,c)

   nodeMonitor = localActivity(n)
   activity(p)
   activity(c)
}

producer(a!) {
   #PRISM
   "Producer"
}

consumer(a?) {
   #PRISM
   "Consumer"
}

localActivity(x?) {
   #REWARD
   x : 1;
}

activity(x?) {
   #REWARD
   "activity"
   x : 1;
}
```

Fig. 12. Reo network with producer, consumer, two FIFO channels and reward monitors

```
mdp

// constant for the upper bound on the data domain of connector variables
const N = 1;

// ---- included from producerConsumer-library.prism --------
module Producer as template
  broken : bool init false;

  ]a[ !broken -> 0.1:(broken'=true) + 0.9:(broken'=false);
endmodule

module Consumer as template
  broken : bool init false;

  ]a[ !broken -> 0.1:(broken'=true) + 0.9:(broken'=false);
endmodule
// ---------------------------------------

module prod = Producer (varprefix=prod_) [ a = p ] endmodule
module cons = Consumer (varprefix=cons_) [ a = c ] endmodule

rewards "nodeMonitor"
 ]n[ true : 1;
endrewards

rewards "activity"
 ]p[ true : 1;
 ]c[ true : 1;
endrewards

module f1
    f1_m : [0..N] init 0;

    ]p[ (f1_m = 0 & !(1 = 0)) & true -> (1) : (f1_m' = 1);
    ]n[ !(f1_m = 0) & 1 = f1_m -> (1) : (f1_m' = 0);
endmodule

module f2
    f2_m : [0..N] init 0;

    ]n[ (f2_m = 0 & !(1 = 0)) & true -> (1) : (f2_m' = 1);
    ]c[ !(f2_m = 0) & 1 = f2_m -> (1) : (f2_m' = 0);
endmodule
```

Fig. 13. Generated PRISM language model description for the REO network with producer, consumer, two FIFO channels and reward monitors