Software and Systems Modeling
https://doi.org/10.1007/s10270-021-00916-8

REGULAR PAPER

Probabilistic modelling and verification using RoboChart and PRISM

Kangfeng Ye[1] · Ana Cavalcanti[1] · Simon Foster[1] · Alvaro Miyazawa[1] · Jim Woodcock[1]

Received: 25 November 2020 / Revised: 15 July 2021 / Accepted: 20 July 2021

Abstract

RoboChart is a timed domain-specific language for robotics, distinctive in its support for automated verification by model checking and theorem proving. Since uncertainty is an essential part of robotic systems, we present here an extension to RoboChart to model uncertainty using probabilism. The extension enriches RoboChart state machines with probability through a new construct: probabilistic junctions as the source of transitions with a probability value. RoboChart has an accompanying tool, called RoboTool, for modelling and verification of functional and real-time behaviour. We present here also an automatic technique, implemented in RoboTool, to transform a RoboChart model into a PRISM model for verification. We have extended the property language of RoboTool so that probabilistic properties expressed in temporal logic can be written using controlled natural language.

Keywords State machines · Formal semantics · Model transformation · PRISM · Probabilistic model checking · Domain-specific language for robotics

1 Introduction

Autonomous robots must carry out their missions without human intervention. Uncertainty in real-world environments, including the physical robotic platform, presents major challenges for these robots. To address these challenges and deal with unknown aspects of the environment, robots often use probabilistic control algorithms. For example, a robot without a detailed map of its environment might resort to using a random walk to carry out its mission, overcoming the uncertainty about its position. This paper addresses modelling and formal verification of robotic controllers that use probabilistic algorithms to deal with uncertainty.

Robotic applications are often designed using state machines that have no formal semantics or even precise syntax. These machines can involve advanced features, including probability, real-time, and shared-variable concurrency. To model and analyse such complex systems, roboticists require knowledge of formal languages and probability theory. Our approach is to use a domain-specific notation, RoboChart [1], and model transformation to support probabilistic reasoning using a notation familiar to roboticists. Usability and automation are key considerations for RoboChart.

Previous work [2] presents the RoboChart, metamodel and semantics, and an Eclipse-based tool, RoboTool, for modelling, verification, and code generation using RoboChart. This paper covers its extension to cater for probabilistic modelling and verification: we have (1) extended RoboChart's state machines to enrich them with a construct for probabilistic choice (with our extension, RoboChart allows four forms of choice between behaviour: conditional, external or guarded by events, nondeterministic, and probabilistic), and (2) developed an automatic technique for verification by model checking using PRISM [3]. Our results are illustrated via several case studies.

The core of RoboChart is a subset of UML state machines that allows modelling of robotic applications, and has a con-

Communicated by Jeff Gray.

✉ Kangfeng Ye
Kangfeng.Ye@york.ac.uk

Ana Cavalcanti
Ana.Cavalcanti@york.ac.uk

Simon Foster
Simon.Foster@york.ac.uk

Alvaro Miyazawa
Alvaro.Miyazawa@york.ac.uk

Jim Woodcock
Jim.Woodcock@york.ac.uk

[1] Department of Computer Science, University of York, York YO10 5GH, UK

Published online: 03 October 2021

strained semantics for automated reasoning. RoboChart can be regarded as a profile of UML, enriched with time constructs. In a RoboChart model, physical robots are abstracted into robotic platforms defined by variables, events, and operations. RoboChart also has a component model with notions of controller and module to foster reuse. A state machine is the basic element to model behaviour. A controller includes one or more parallel state machines. Communication between state machines in the same controller is synchronous, while communication between controllers can also be asynchronous. A module defines the boundaries of the robotic application and is composed of one robotic platform and one or more controllers.

RoboChart has a semantics for formal reasoning based on Hoare and He's Unifying Theories of Programming (UTP) [4], with the process algebra CSP [5–7], which has a UTP semantics [4,7], used as a front-end. RoboTool generates models written in CSP and tock-CSP [6,8], an encoding that uses an event *tock* to mark the passage of time, from RoboChart models [2] to enable use of the refinement model checker FDR [9,10] to verify properties, such as deadlock and livelock freedom. We can also verify specific behavioural properties, including time budgets and deadlines. We cannot, however, verify probabilistic properties.

Standard CSP and tock-CSP do not support probabilistic choice. An experimental extension of CSP included a probabilistic choice construct, and a specific version of FDR was developed to translate refinement checks in CSP to the PRISM language [3] through the WatchDog Transformation [11]. This approach, however, only supports CSP traces refinement reasoning [6] (so limited to reachability properties) and is not supported by recent FDR versions.

RoboChart also has support for semiautomated verification by theorem proving using Isabelle/UTP [12,13]. Both FDR and Isabelle/UTP verification target only proof of reactive and timed behaviours of RoboChart models, not probabilistic, as we do here.

RoboChart differs from the PRISM language in several aspects: abstraction level, data types, variable sharing, and event synchronisation. Our technique to generate PRISM scripts addresses all these issues. The technique is formalised by transformation rules that we present here, and automated in RoboTool.

A property language facilitates the writing of properties for verification in PRISM using a controlled natural language. It can express probabilistic properties that are based on temporal logic. This also improves performance, by enabling the running of multiple PRISM instances in parallel when several properties are stated.

In [2], we have described a CSP semantics for a version of RoboChart that supports standard features of statecharts and time constructs, but not probabilistic choice. In [14], we have studied the imperative, sequential action language for RoboChart, and explicated the weakest completion approach [15] to give UTP semantics to a nondeterministic probabilistic programming language. This work shows how informal proofs can be turned into formal proofs for implementation in Isabelle/UTP, which enables future application of theorem proving to verify RoboChart models. Here we cover most features of RoboChart, not only its imperative and sequential action language, and pursue support for model checking, not theorem proving.

A recent work by Conserva Filho et al. [16] interprets the probabilistic choice in RoboChart using the probabilistic CSP operator ${}_{p}\boxplus_{1-p}$ [11]. This preserves semantics for all other constructors in RoboChart, because RoboChart's semantics is given in CSP. Verification uses refinement model checking: $S \sqsubseteq_T I$. Here, S is the property to be verified and described by a nonprobabilistic CSP process, and I is a probabilistic CSP process under analysis, generated from a probabilistic model in RoboChart. An extended version of FDR supports probabilistic choice in CSP and translation of the refinement check to a PRISM model. Our work is different from that in [16] in several aspects. Verification in [16] covers only trace refinement and so is limited to reachability properties. Our work supports all the temporal logics in PRISM. CSP processes are used in [16] to specify properties. Consequently, users need to have knowledge of CSP. In contrast, in our work, we use a customised property language, RoboCert. Finally, their approach is only supported by one version of FDR and not by the more recent versions being used with RoboTool.

Our novel contributions here are as follows: (1) the introduction of a new construct for probabilistic choice in RoboChart's state machines for probabilistic modelling; (2) a metamodel for PRISM; (3) RoboChart's probabilistic semantics in PRISM (for a subset of RoboChart constructs), including not only the new probabilistic construct but state machines and the component model in a context where probabilities are captured; (4) the implementation of the semantics in RoboTool for automated generation of PRISM models; and (5) a property language for verification of qualitative and quantitative properties.

The remainder of this paper is organised as follows. We review related work in Sect. 2. Section 3 examines core features of RoboChart, introduces our new probabilistic choice construct through an internal mail delivery robot example, and presents extra well-formedness conditions. In Sect. 4, we describe the PRISM language and its semantics, and present a PRISM metamodel that we have developed for transformation. Section 5 formalises our technique to transform RoboChart into PRISM models, and illustrates it using the mail delivery robot. Section 6 describes how RoboTool supports probabilistic modelling and the application of the transformations to automatically generate PRISM models. The property language for probabilistic properties is described in Sect. 7. Finally, we conclude and discuss future work in Sect. 8.

2 Related work

In this section, we set RoboChart in context by discussing its relation to fundamental probabilistic frameworks or models in Sect. 2.1, notations for probabilistic modelling with rich expressions and high abstraction in Sect. 2.2, and model-based domain-specific languages for robotics in Sect. 2.3.

2.1 Fundamental probabilistic frameworks

This section describes notations and formalisms that are at the same level of abstraction as PRISM, but at a different level of abstraction as RoboChart. Some of their features, however, have inspired the design of RoboChart as acknowledged below.

Segala and Lynch [17] introduce probabilistic automata as a specialised form of labelled transition system. A transition is labelled with probability values, so that they map a source state to probability distributions over (action, target state) pairs. Actions can be external, modelling interactions with the environment through events, or internal, modelling computation steps through internal events τ. If, in each probability distribution, the action is always the same for every target state, the probabilistic automaton is called simple. Two simple probabilistic automata can be composed in parallel; this is a kind of combination that is also available in RoboChart, via use of multiple machines and controllers in a model.

The probabilistic semantics of RoboChart that is embedded in PRISM, as presented later on, can be regarded as corresponding to a simple probabilistic automaton in that the actions in a probability distribution for every target state are the same. Transitions map states and actions to probability distributions over states. RoboChart also distinguishes nondeterministic choice and probabilistic choice. RoboChart, however, has more modelling constructs and is more expressive, when compared to probabilistic automata.

Hansson [18] presents an alternating model that distinguishes between nondeterministic and probabilistic choice. Either a nondeterministic choice or a probabilistic choice can be made in each state of this model, and the order of availability of these choices is strictly alternating between a nondeterministic and a probabilistic choice. For the target state of a transition originating from a nondeterministic state (that is the state where only a nondeterministic choice can be made), only a probabilistic choice can be made. Conversely, for the target state of a transition originating from a probabilistic state (that is the state where only a probabilistic choice can be made), only a nondeterministic choice can be made. RoboChart adopts the same alternation between nondeterministic and probabilistic choice, but the point where a probabilistic choice is made in RoboChart is not a state. Instead, a probabilistic choice is made within a transition originating from a nondeterministic state. RoboChart is similar to a simple probabilistic automaton in this aspect. The alternating model and the simple probabilistic automata model with respect to probabilistic bisimulation (strong bisimulation) [19] are isomorphic [20]: one model can be translated to another, and vice versa.

Van Glabbeek et al.'s reactive model [21] partially corresponds to the simple probabilistic automaton model as it allows external nondeterministic choice between different actions (that is, multiple transitions with different external actions from the same state), but without internal nondeterministic choice involving the same actions. In RoboChart, on the contrary, we can model internal nondeterministic choice via transitions that have no trigger, or transitions from the same state with the same trigger.

A variety of Markov models are used for discrete probability modelling. Discrete-time Markov chains (DTMCs) [22,23], in which each state leads to a probabilistic choice directly, are purely probabilistic without nondeterminism. A DTMC is equivalent to a fully probabilistic automaton in discrete time.

Markov decision processes (MDPs) [24,25], which extend DTMCs, can be regarded as simple probabilistic automata without internal nondeterministic choices. From each state, an action is associated with a probability distribution. Transitions from the same state with the same actions are, therefore, not allowed. Similar to MDPs, RoboChart models exhibit both probabilistic and nondeterministic choice.

A variety of DTMCs and MDP-based languages have been designed to facilitate systems modelling and analysis. Additionally, labelling functions for states (called atomic propositions) and cost or reward functions for states and transitions are also introduced. The PRISM language, which is supported by the widely used probabilistic model checker PRISM and other model checking tools such as Storm [26], is one such language. It is based on reactive modules [27] and is a low-level guarded-command language.

MODEST [28,29], a modelling and analysis framework for stochastic hybrid systems, uses a comparatively higher-level language that is inspired by process algebras. It is supported by the MODEST Toolset.[1] PRISM, Storm, and MODEST all support DTMC and MDP models for discrete probabilities.

These languages often are textual and support only basic types of variables: boolean, integer numbers, and real numbers. To use the tools, knowledge of underlying probability models, formal methods, and temporal logic is often required.

[1] www.modestchecker.net/.

On the contrary, RoboChart is a diagrammatic notation, in line with current practice of developing robotic control software often using diagrammatic state machines [30–33]. RoboChart offers more facilities for abstraction. This includes a component model to define services of a platforms, and an architecture of controllers and parallel threads, and their connections. In addition, RoboChart offers a richer set of data types, including many mathematical data types such as relations and functions, in addition to basic data types, to facilitate reasoning. The comprehensive expression language of RoboChart is that of the Z notation (see the ISO Z Standard[2]), albeit with a more roboticist-friendly syntax and a few syntactic-sugaring constructs.

Table 1 shows a comparison between these probabilistic frameworks and RoboChart according to their support of nondeterministic choice and probabilistic choice, abstraction, and expressiveness. Those frameworks provide different mathematical accounts of probabilistic behaviour, and have a role similar to that of PRISM, rather than RoboChart, in our work.

2.2 Improved modelling languages

A probabilistic finite state machine (PFSM) [33] extends state machines with probability and real-time behaviour. It is used to describe the behaviour of an individual foraging robot in a swarm. Both simulation [33] and formal verification [34] are used to understand and analyse the behaviour of the controller. A PFSM guides the development of a simulation and a formal model, but only loose connections between the machine and the simulation or the model are claimed. RoboChart also extends state machines with probability and real-time behaviour, but it has a formal semantics that can be automatically generated for given models.

Probabilistic timed Behaviour Trees (ptBTs) [35] extend Behaviour Trees [36], a formal and graphical notation to construct a design out of a set of functional requirements in a stepwise and traceable way, with probabilistic behaviour. The meaning of a ptBT is given using probabilistic timed automata (PTAs) [37]. Verification of a ptBT model is realised through translation to a DTMC or MDP model, which is analysed using PRISM. Notations based on behaviour trees treat an individual functional requirement as a behaviour tree, and integrate all these trees into an integrated design behaviour tree that allows defect detection and is traceable to the requirements. Compared to the notations based on state machines, those based on behaviour trees are not well developed and studied. One reason is the lack of sophisticated tool support.

Among the modelling notations extended to accommodate probability, UML state machines [38] are the most popular. DAMRTS (Dependability Analysis Models for Real-Time Systems) [39,40] is an UML profile that extends UML Statecharts with time and probability. Discrete probabilities are associated with events and a discrete probability distribution is defined over (event, state) pairs. We need a construct to group all transitions that are from the same state and form a probability distribution, but it is not clear what method, if any, is used for that in DAMRTS. From an example in [39], transitions with probabilities do not form a complete cover. So it can be the case that a transition is labelled with an event and a probability p, but there are no other transitions with the same event and the complementary probability $1 - p$. The semantics in this case is not clear. DAMRTS models can be automatically translated to probabilistic timed automata and analysed by PRISM. This translation is, however, not formalised. In contrast, in RoboChart, a probability distribution is associated with a single transition, which defines the target states like in a probabilistic automaton. Moreover, the probabilities in such a transition add up to 1. So, there is no ambiguity. Our translation to PRISM is formalised in this paper.

Jansen et al. [41] describe a probabilistic conservative extension of UML's Statechart notation, called P-statecharts. Since the extension is conservative, we know that all the behaviours of the original semantics are behaviours of the probabilistic semantics. A transition can lead to one of several states depending on a specified probability distribution. Each probability distribution is guarded by a trigger, either an event or no event, which corresponds to an external action or an internal action τ in a probabilistic automaton. Discrete probabilities are associated with actions and a discrete probability distribution is defined over (action, state) pairs. A P-statechart alternates between nondeterministic choices and probabilistic choices using the strictly alternating model [18,42]. Transitions that exit nondeterministic choices enter only probabilistic choices. A P-statechart deals not only with nondeterminism and probabilism, but also with priorities within Statecharts. The semantic models of P-statechart are based on MDPs. Properties of a P-statechart written in the probabilistic branching time logic (PCTL) can, therefore, be automatically checked using PRISM.

Probabilistic modelling and verification using RoboChart are inspired by Jansen et al.'s P-statecharts. Similar to P-statecharts, RoboChart also deals with nondeterminism and probabilistic choices in the same strictly alternating way: transitions that exit nondeterministic choices enter only probabilistic choices. In the probabilistic model, even if two states are connected by a transition directly, therefore without an explicit probabilistic choice, it is treated as having an implicit probabilistic choice with the probability of the outgoing transition as 1. So, since every transition is prob-

[2] http://standards.iso.org/ittf/PubliclyAvailableStandards/c021573_ISO_IEC_13568_2002(E).zip.), albeit with a more roboticist-friendly syntax and a few syntactic-sugaring constructs.

Table 1 Comparison of related work (fundamental probabilistic frameworks or models)

Work	Nondeterministic choice		Probabilistic choice	Types/expressiveness
	External	Internal		
Probabilistic automata	Yes	Yes	Yes	N/A
Alternating model	Yes	Yes	Yes	N/A
Reactive model	Yes	No	Yes	N/A
PRISM	Yes	No	Yes	Basic
Storm	Yes	No	Yes	Basic
MODEST	Yes	No	Yes	Basic
RoboChart	Yes	Yes	Yes	Rich

Basic types/expressiveness: no support for structured data types. Rich types/expressiveness: some support for data types such as sets, sequences, and so on

abilistic, after a nondeterministic choice selects an enabled transition, there is a probabilistic choice.

A conflict arises when a transition from a composite state and a transition from a substate are both enabled. A P-statechart assumes a given priority scheme to resolve the conflict. RoboChart has a simpler semantics, which leaves the choice nondeterministically. In this way, we treat them as another level of nondeterminism in order to avoid dealing with the complexity of priorities in the semantics. Differently from P-statecharts, RoboChart does not support AND-states within a state machine (but supports parallelism between state machines) to simplify the semantics, and, therefore, make reasoning more tractable and improve automation of verification. Regarding triggers, RoboChart supports input and output triggers in addition to simple triggers just for synchronisation.

Richer action constructs are available in RoboChart, including assignments, communications, sequential composition, and conditionals. P-statecharts only support assignments and communications. Functions and operations can be specified using preconditions and postconditions described in the rich language of Z predicates.

Table 2 summarises the comparison of RoboChart with the modelling languages above.

2.3 Domain-specific languages for robotics

Model-driven software development in robotics has been studied [43] and several domain specific languages are available [44]. SmartSoft[3] [45], V^{3}CMM [46], BRICS Component Model (BCM) [47], RobotML [48], SafeRobots [49], and the work in [50] all employ component models like in RoboChart. Although some of them, such as BCM, have a degree of formal modelling, they usually do not have a formal semantics (and, therefore, cannot support formal reasoning and verification) or have a formal semantics only for part of their constructs. RoboChart, however, has a formal semantics defined for all its constructs, including its component model.

GenoM3 [51,52] is a model-based engineering framework for robotics software. It supports verification using model checking via translation to Petri Nets [53] and deadlock checking using BIP [54]. It is also an executable language. RoboChart, nevertheless, provides various levels of abstraction, and also supports theorem proving in addition to model checking.

As far as we know, none of these notations support probabilistic modelling and reasoning, as we propose here for RoboChart. Extending these notations, however, may benefit from the results we present here.

Thrun et al. [55] uses MDPs as the underlying mathematical framework to model uncertainty in robot action selection for probabilistic planning and control, and introduces the value iteration algorithm to find control policies for these models. They use partially observable Markov decision processes (POMDPs) to model uncertainty in perception because the environment is usually only partially observable through sensors. Probabilistic modelling in RoboChart, as discussed in this paper, covers uncertainty in robot action selection. Our probabilistic semantics is also based on MDPs. RoboChart, however, supports different levels of abstraction through a component model, abstract data types, and a rich action languages (see Table 2).

In summary, the probabilistic modelling of RoboChart is based on existing notations, but provides abstraction and improved modelling practice for the robotics domain. The analysis of probabilistic behaviour in RoboChart models is fully automated, and the transformation from RoboChart to PRISM is formalised.

Next, we describe and illustrate RoboChart.

[3] smart-robotics.sourceforge.net/.

Table 2 Comparison of related work (improved modelling languages)

Work	Abstraction				Formal semantics	Tool support	Note
	Component model	Types	Action constructs	Refinement			
PFSM	No	N/A	N/A	No	No	No	Informal description of behaviour
ptBTs	No	Basic	Basic	No	PTA	Model checking	Manual translation
DAMRTS	No	Basic	Basic	No	PTA	Model checking	Automated translation (informal)
P-statecharts	No	Basic	Basic	No	MDP	Model checking	
RoboChart	Yes	Rich	Rich	Yes	UTP (CSP) and MDP	Model checking Theorem Proving	Automated verification

Basic actions: only primitive actions (assignments, synchronisations, and so on) and no control-flow constructs. Rich actions: some control-flow constructs (conditionals, sequences, and so on)

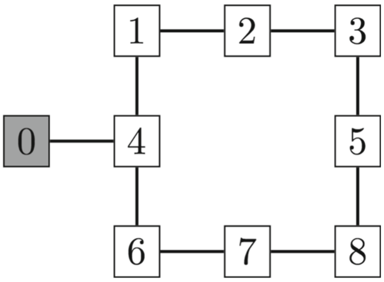


Fig. 1 Map of the workplace from [56]

3 RoboChart

This section gives an overview of our approach to probabilistic modelling using an example. Part of the novelty of our work is related to the component model of RoboChart, not only the novel state machine constructs. For this reason, we present an example that illustrates the use of probabilistic choices, challenging constructs of machines, such as composite states, and the core elements of the RoboChart component model: controllers, robotic platforms, and modules. In Sect. 3.1, we present our probabilistic choice operator and an informal account of its effect on the RoboChart semantics. We refer to the RoboChart reference manual [1] for a complete account of the notation. Sections 3.2 and 3.3 present the metamodel and well-formedness conditions for the probabilistic constructors.

3.1 Notation

We describe our facilities for modelling probabilistic controllers using as an example a mail delivery robot from [56]. It delivers mail to eight offices arranged in the configuration in Fig. 1. An office 0 is the station to charge its battery. A worker in an office can send mail to another, but not to the charging station.

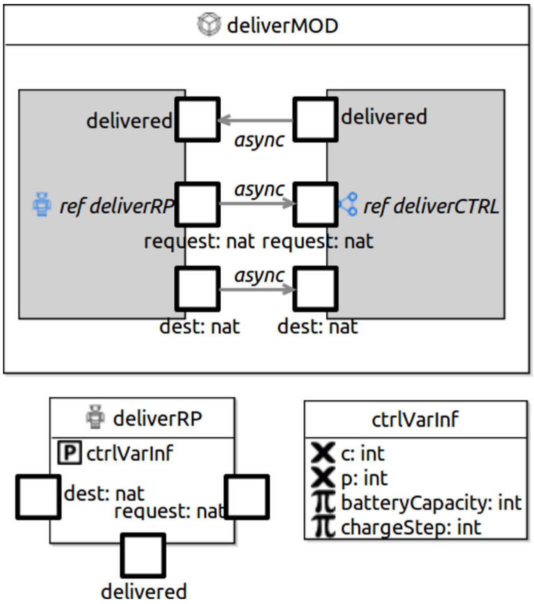


Fig. 2 Module and Robotic platform in RoboChart model for delivery robot

The robot can deal with only one delivery at a time. Upon receipt of a delivery request, the robot starts to fetch the mail by moving to the sending office. Upon arrival, it fetches the mail and then moves to its destination to deliver it. After delivering the mail to its destination, it is ready for the next request.

The robot is not equipped with particular cameras or sensors for dynamic route allocation. Instead, it uses a random algorithm, similar to those of vacuum cleaning robots, to choose its moves. With equal probability, it either stays where it is or moves to adjacent offices. For example, if its current location is at office 1, it remains at office 1, moves to office 2, or moves to office 4. Each choice has probability 1/3.

Module and Robotic Platform. The module of our RoboChart model for this example is shown in Fig. 2. The module deliv-

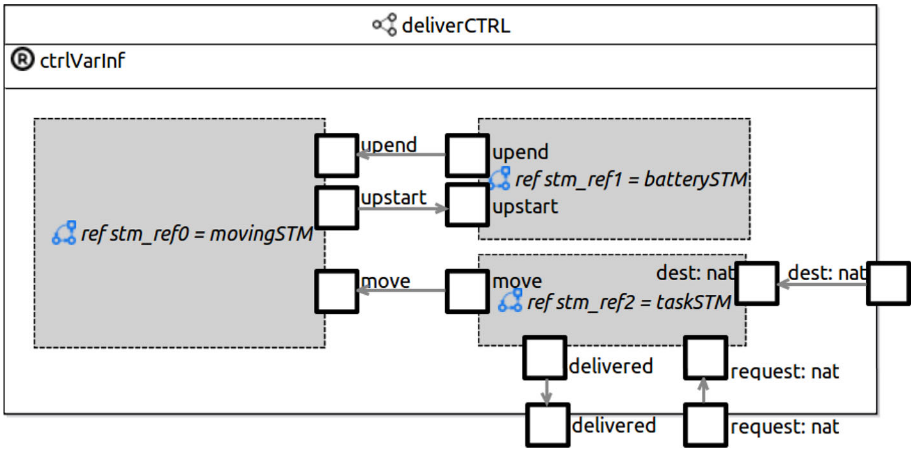


Fig. 3 Controller in RoboChart model for delivery robot

erMOD is composed of one robotic platform deliverRP and a controller deliverCTRL.

The notion of robotic platform is a crucial element in the definition of a module (and so, of a model). It captures the services that need to be offered by a robotic platform to enable the use of the control software. For abstraction, these services are not specified in a module. This ensures that a module is platform-independent. The robotic platform, however, is the link to connect a module to a model for a physical robot, for example. This is described in detail in [57].

A platform includes variables, events, and operations that represent sensors and actuators. It may also define constants that characterise parameters of the model. In our example, we collect in an interface ctrlVarInf variables and constants of deliverRP. The current location and battery level of the robot are recorded as variables ✖ p and c. The battery capacity batteryCapacity and the battery charged per update chargeStep are design parameters and treated as constants π.

We use interfaces to group elements for reuse and to describe dependencies. The interface ctrlVarInf in Fig. 2 is provided 🄿 by the robotic platform. The variables and constants defined in ctrlVarInf are used in the controller and all its state machines, which require Ⓡ the interface (see Figs. 3, 4, 5, and 6).

In addition to variables and constants, the robotic platform can communicate with controllers via connected events □.

The controller deliverCTRL in Fig. 2 gets a mail delivery request including an office number that locates the mail to be delivered and a destination office number from the platform via the typed events request and dest. After delivery, the controller notifies the robotic platform via the delivered event.

Controllers. The definition of the deliverCTRL controller is given in Fig. 3. It has three state machines ☍: movingSTM, batterySTM, and taskSTM, which correspond to three separate functionalities: movement control, battery management, and task management. The machines are defined in Figs. 4, 5, and 6.

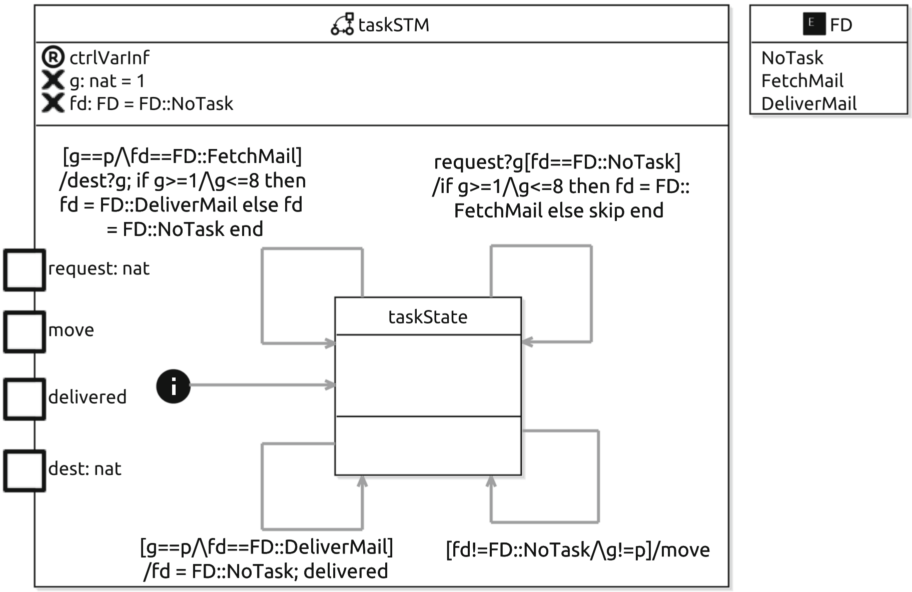
A controller ⋘ in RoboChart encapsulates one or more (parallel) state machines. Connections between a controller and its state machines are used to relay information through the events of the controller. The connections on the events request and dest of the controller in Fig. 3 pass information from the platform to the machine taskSTM. The machine taskSTM acknowledges delivery via the delivered event relayed through the event delivered of the controller.

Connections between machines represent synchronisations and interactions. The machines movingSTM and taskSTM shown in Fig. 3 synchronise with each other on a move event used to trigger movement when the current position of the robot is different from the goal (either the mail request office or mail destination office), while movingSTM and batterySTM synchronise on the upstart and upend events to trigger start and end of an update of the current battery level after a move.

State machines. Behaviour descriptions are given by state machines in RoboChart. They may require interfaces and declare local constants, variables, and events, like controllers. For example, the taskSTM machine (Fig. 4) declares two variables g and fd. The variable g is a natural number: an office number where mail needs to be fetched or delivered; g is initialised to 0. We call g the goal of the robot. The variable fd is an element of the enumeration FD defined in Fig. 4. It represents three stages of a delivery task: idle, fetching, and delivering mail. The variable fd is initialised to NoTask. These variables are local to taskSTM. All declared events appear on the border of machines.

A state machine comprises states, junctions, and transitions. States may be simple or composite. A composite state itself contains a state machine. A state may also have entry, exit, and during actions, executed when the state is entered, exited, and active.

Fig. 4 Task state machine


taskSTM
ctrlVarInf
g: nat = 1
fd: FD = FD::NoTask
FD
NoTask
FetchMail
DeliverMail
[g==p/\fd==FD::FetchMail] /dest?g; if g>=1/\g<=8 then fd = FD::DeliverMail else fd = FD::NoTask end
request?g[fd==FD::NoTask] /if g>=1/\g<=8 then fd = FD:: FetchMail else skip end
request: nat
move
delivered
dest: nat
taskState
[g==p/\fd==FD::DeliverMail] /fd = FD::NoTask; delivered
[fd!=FD::NoTask/\g!=p]/move

Transitions connect states and junctions. A self-transition has the same source and target states. Transitions have a label with the following optional features:

- A *trigger* event;
- A *guard*, specifying the conditions that need to hold for the transition to be enabled;
- A *probability value*, defining the probability of occurrence of this transition; and
- An *action* that is executed if the transition is taken.

Actions are statements. The action constructs are skip, an action that terminates immediately, assignment (=), sequential composition (;), conditional (if), input events (of the form evt?v where evt is an event and v is a variable that records the input value), output events (of the form evt!e where e is an expression whose value is output), or synchronisations (of the form evt or evt.e).

A junction is different from a state in that the control flow of a machine cannot stop at a junction. With a junction, we can break the flow of a transition by creating points of decision in between states. Junction is a concept of UML state machines and other Statechart notations. Junctions represent decision points that must be made immediately. As a consequence, a junction must always have at least one outgoing transition enabled in order to leave the junction. RoboChart has three types of junctions: initial junctions ⓘ, normal junctions ●, and probabilistic junctions Ⓟ defined here. An initial junction indicates the starting point of execution of a state machine, and cannot have incoming transitions. Normal junctions have both incoming and outgoing transitions, but these transitions cannot be labelled with probability values.

Probabilistic junctions, also used in other techniques [58,59], are a special form of junction introduced to capture probabilistic choices. Only transitions from a probabilistic junction can be labelled with probability values, which must add up to 1. This imposes a proof obligation, since probabilities can be given by expressions involving variables.

The taskSTM machine in Fig. 4 has just one state taskState with four self-transitions. The top-right transition is taken when there is a mail delivery request (trigger request?g) and the robot is idle (guard fd=FD::NoTask where FD::NoTask is a constant of the enumeration FD). If that transition is taken, its action establishes that if the goal g is an office number from 1 to 8, the robot moves to the fetching mail stage (FetchMail); otherwise, it ignores the request (skip). The bottom-right transition is taken if the robot is not idle and its current position is not the goal. If so, the machine waits for a synchronisation on the move event, as specified by the transition action move (a synchronisation action). The top-left transition is enabled if the robot arrives at the office to fetch mail (g==p) when it is fetching mail. If that transition is taken, the machine waits for an input dest?g from the platform giving the delivery destination g. If the destination office is valid (between 1 and 8), the robot switches to the delivering stage. Otherwise, it discards the request. The bottom-left transition is enabled if the robot reaches the destination when it is delivering. If so, it returns to the idle stage and sends a delivered acknowledgement.

The batterySTM machine in Fig. 5 manages the battery. It has only the state batteryState. The transition from the initial junction sets the battery level c to its capacity batteryCapacity to record that initially the battery is full. Two self-transitions of batteryState can be taken when the event upstart occurs. One is taken when the robot is at the charg-

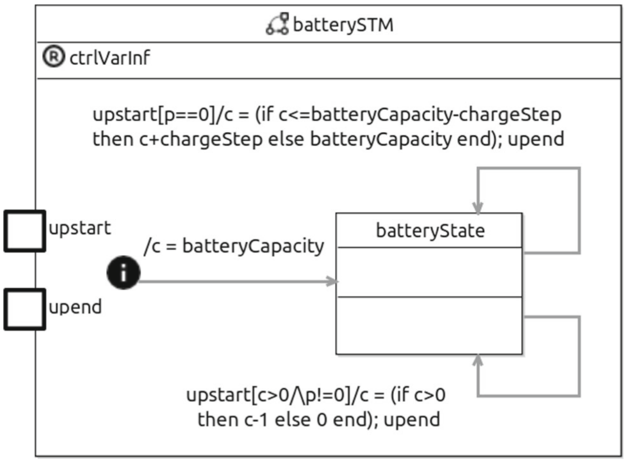

batterySTM
ctrlVarInf
upstart[p==0]/c = (if c<=batteryCapacity-chargeStep
then c+chargeStep else batteryCapacity end); upend
upstart
upend
/c = batteryCapacity
batteryState
upstart[c>0/\p!=0]/c = (if c>0
then c-1 else 0 end); upend

Fig. 5 Battery state machine

ing station (p==0 , where p is provided by deliverRP shown in Fig. 2 and required here through an interface ctrlVarInf), and another when the robot is not there and the battery is not empty. In the first scenario, the battery is charged and its level is increased by the constant chargeStep per update till it is full. In the second scenario, one unit of the battery is consumed. In both scenarios, the upend event is used to signal that the battery state has been updated. Both upstart and upend are synchronisations with events in the movingSTM machine to allow the battery to be updated when the robot is moving.

The movingSTM machine in Fig. 6 controls movement of the robot. The machine has two states: Stuck and Move. The transition from the initial junction sets the current position p of the robot to the charging station. The Move state is composite. Its machine has nine states (from s0 to s8). Each such state corresponds to one office and has an entry action such as p=2 that sets p to 2 upon entering of the state s2 to record in p the office in which the robot is located. The transitions from such states are connected to probabilistic junctions and are labelled with a trigger move and a guard condition (c>0). So, these transitions are taken only when movement is trig-

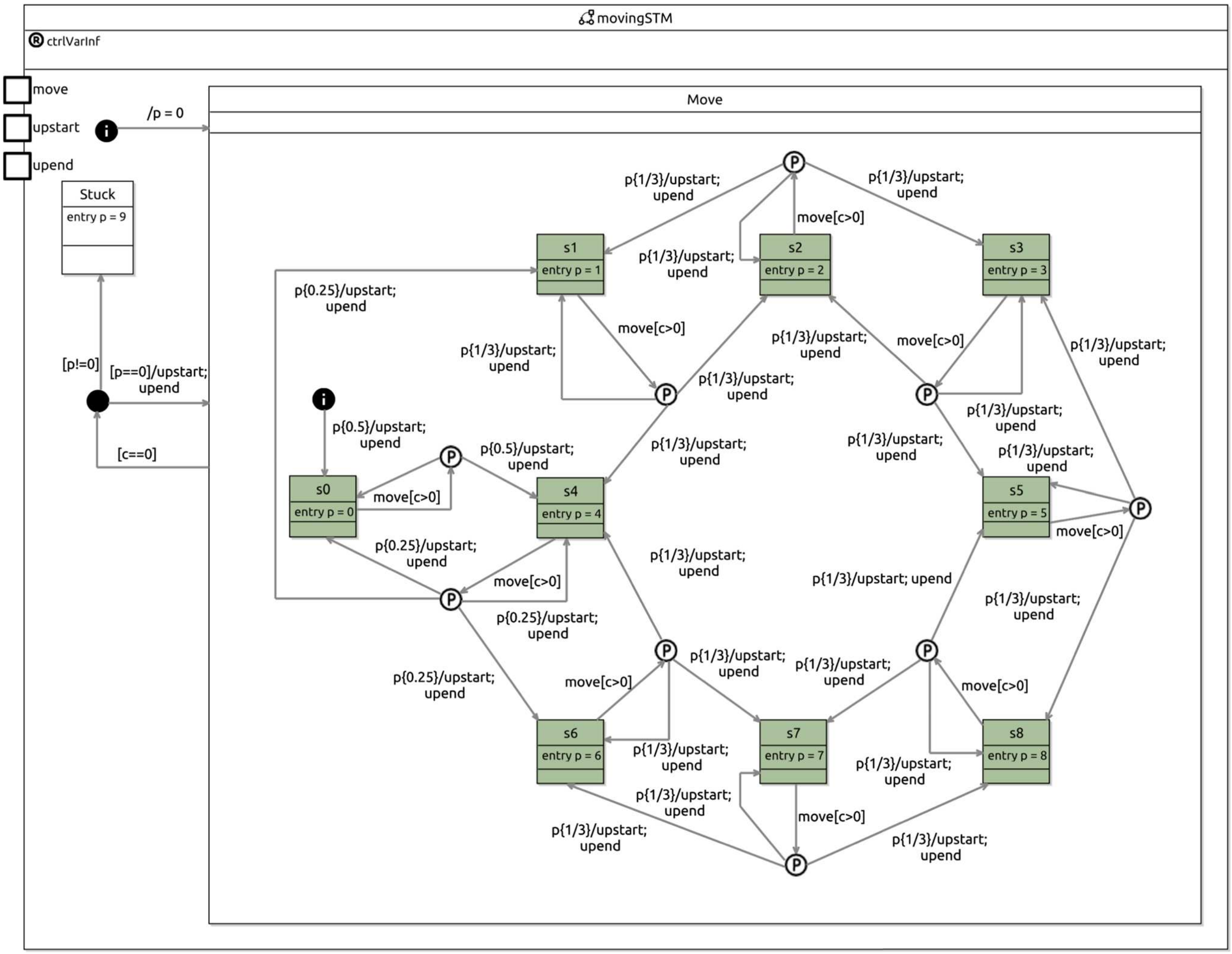


Fig. 6 Movement state machine

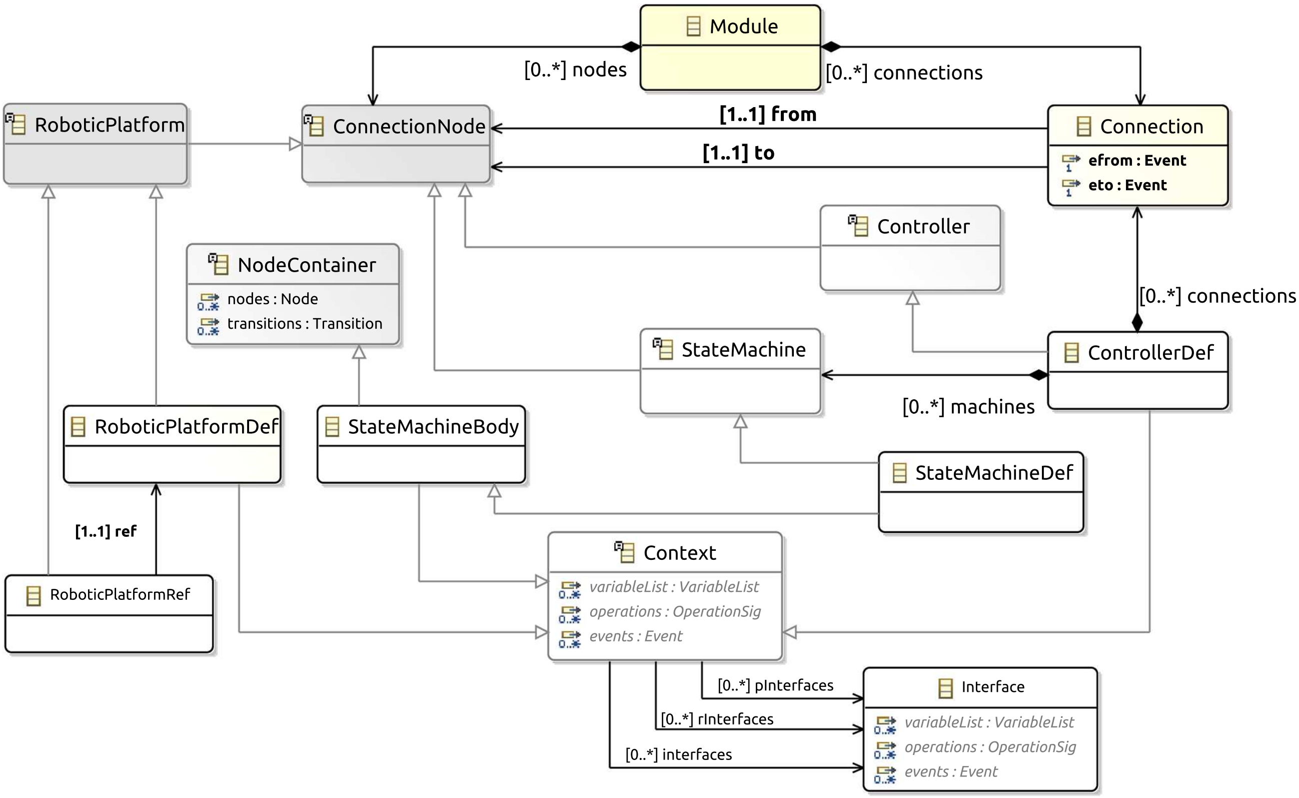


Fig. 7 Metamodel of a RoboChart module

gered by the taskSTM machine and the battery is not empty. All transitions leaving one of the probabilistic junctions are labelled with the same probability value, and so the robot randomly chooses its next position based on the map in Fig. 1. No matter which outgoing transition is taken, the machine updates the battery by synchronising with the batterySTM machine using upstart and upend events.

When the robot is not at the charging station and its battery is empty, the transition from the Move state with guard c==0 is taken. A junction in this transition defines a decision based on the current robot position. If it is at the charging station, the battery is charged by synchronising with the batterySTM machine and movingSTM reenters Move. Otherwise, the machine movingSTM enters the Stuck state because the robot has no battery and cannot charge. Stuck has no outgoing transitions. So the robot cannot move again.

Next, we describe the RoboChart metamodel, and return to this example later in Sect. 5.2.

3.2 Metamodel

Here, we describe the RoboChart metamodel [1] and the changes needed to include probabilistic choice.

A RoboChart model is defined by a module, whose metamodel is shown in Fig. 7. A Module comprises a collection of nodes (ConnectionNode) and connections (Connection). A ConnectionNode can be a RoboticPlatform, a Controller, or a StateMachine. A RoboticPlatform is either given by a definition (RoboticPlatformDef) or by a reference to a definition (RoboticPlatformRef). A RoboticPlatformRef refers to exactly one RoboticPlatformDef. The metamodel for controllers and state machines is similar. A node in a module can be a definition or a reference to a definition of a robotic platform or a controller, but not a state machine as specified by well-formedness conditions discussed later.

A StateMachineDef is a StateMachineBody that can be used as a ConnectionNode (in a ControllerDef). So, StateMachineDef needs to inherit from both StateMachineBody and ConnectionNode. We, however, have an intermediate concept of a StateMachine that inherits from ConnectionNode. StateMachineDef inherits from ConnectionNode via StateMachine, which allows different forms of definition. In particular, a StateMachine may be defined by reference to a StateMachineDef.

Robotic platform and controller definitions may provide and require interfaces through extending Context. The class Context contains variables (VariableList), operations (OperationSig), and events (Event), and refers to three kinds of interfaces: provided interfaces (pInterfaces), required interfaces (rInterfaces), and defined interfaces (interfaces).

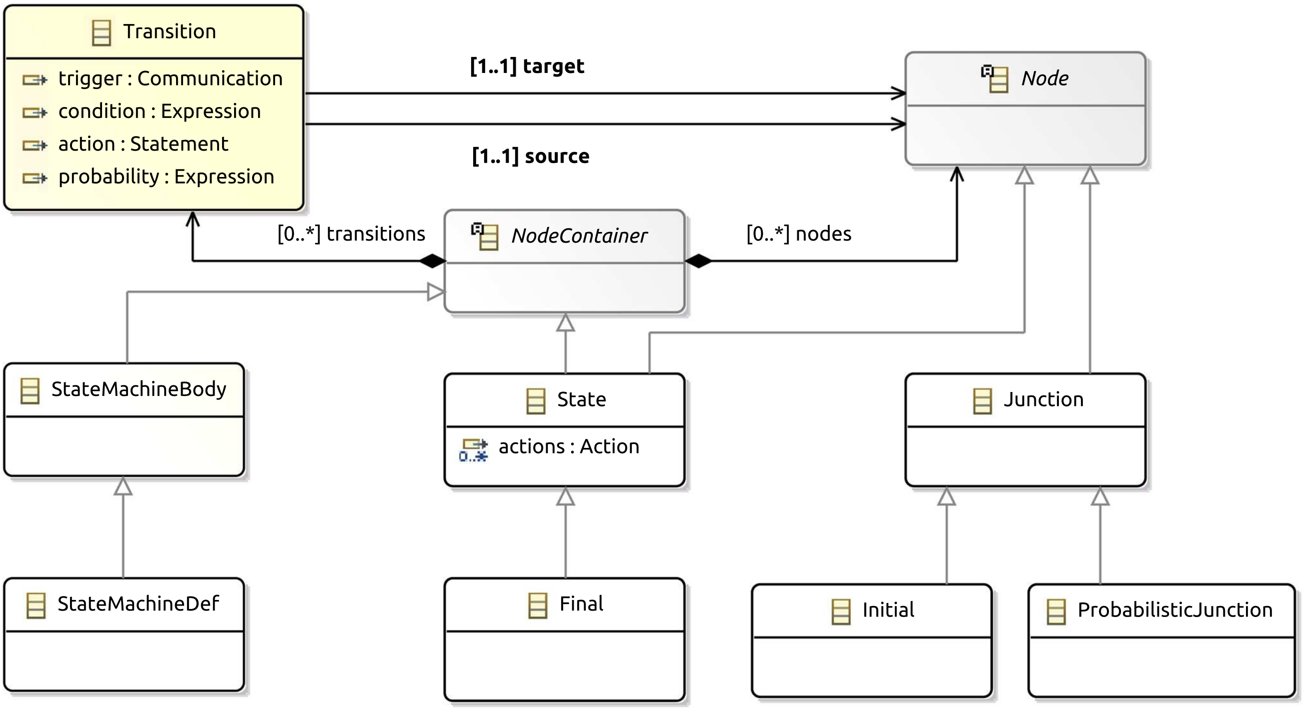


Fig. 8 Metamodel of RoboChart state machines

Connection links one node to another by their events. A connection can be asynchronous or synchronous, and unidirectional or bidirectional. The metamodel permits connections between any two nodes, but not every connection is valid. For example, a robotic platform cannot connect to a state machine. We connect them through a controller. Well-formedness conditions enforce such restrictions.

ControllerDefinitions, are composed of a collection of state machines and connections. The structure of a state machine is detailed in Fig. 8. A state machine is a NodeContainer that includes a collection of nodes (Node) and transitions (Transition). States (State), junctions (Junction), and probabilistic junctions (ProbabilisticJunction) are possible nodes. State is also a NodeContainer, so a state can include nodes and transitions. This supports hierarchical state machines. State contains entry, exit, and during Actions. Initial is a junction and Final is a state. Transitions are directional and connect one source node to a target node. A transition may have a trigger (Communication), a guard (Expression), a probability value (Expression), and an action (Statement).

The metamodel of variables, communications, events, and operations is shown in Fig. 9. A Variable, an OperationSig, and an Event all have a Name. A Variable also has a Type and an optional initial value, given by an Expression. A VariableList contains one or more variables and has a modifier (VariableModifier) from which the counterpart in a Variable derives. An operation signature (OperationSig) contains zero or more parameters (Parameter). A Communication has a type (CommunicationType), and may have a parameter (Variable), a value (Expression), and refer to an Event. An Action contains exactly one Statement. A CommunicationStmt is a Statement that contains exactly one communication: that uses an Event.

The changes we have made to the original RoboChart metamodel presented in [1] to cater for probability are minor: addition of ProbabilisticJunction, and a probability value to Transition. Next, we define healthiness conditions associated with these constructs.

3.3 Well-formedness conditions

Well-formedness conditions define restrictions on models to make them meaningful. It includes typing and scope rules for expressions and actions, and imposes various conditions, such as uniqueness of names in all components. The RoboChart reference manual [1] gives a full account of these conditions.

We now present the extra well-formedness conditions imposed on transitions (Sect. 3.3.1) and on the new probabilistic junction (Sect. 3.3.2).

3.3.1 → Transitions

PT1 The source of a transition with a probability value must be a probabilistic junction, that is, states, initial junc-

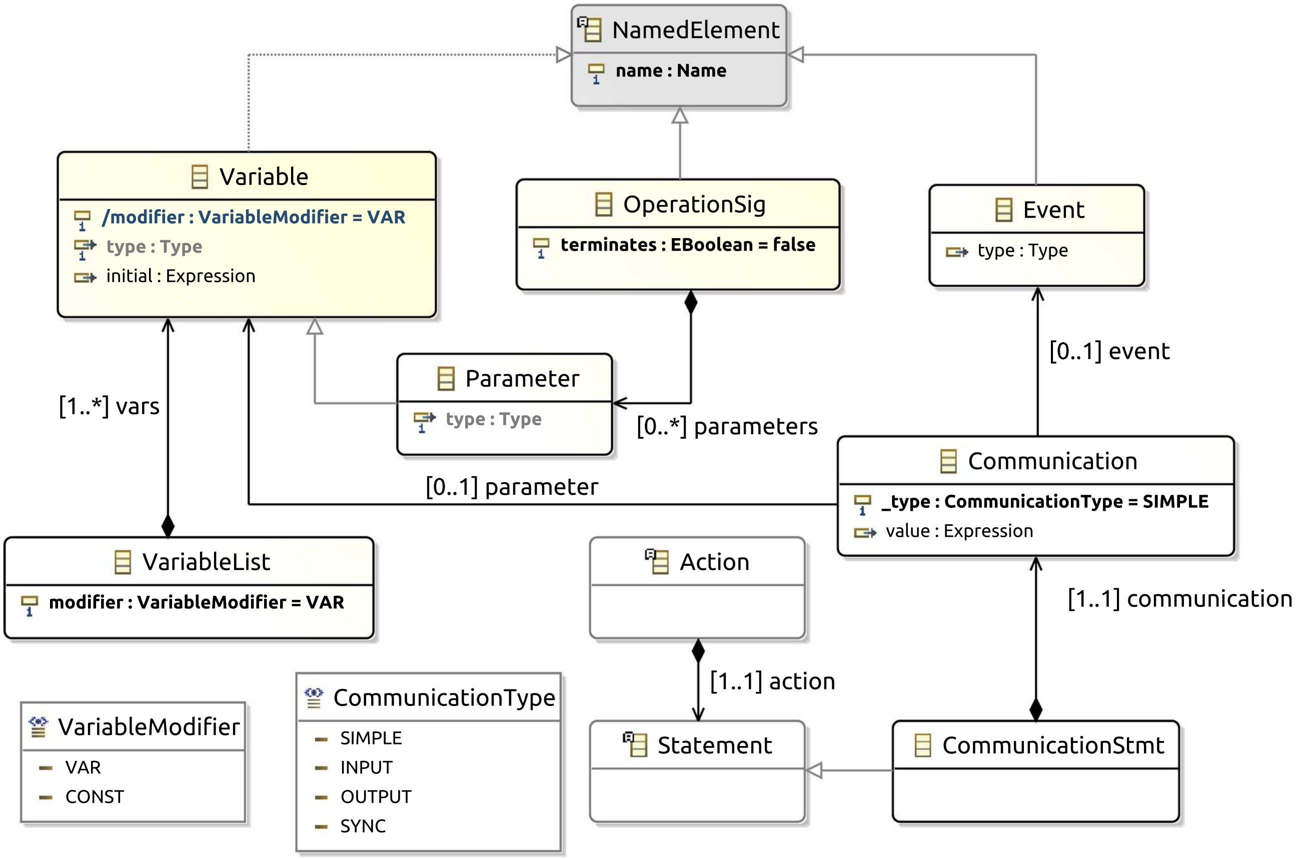


Fig. 9 Metamodel of RoboChart variables, communications, events, and operation signatures

tions, and normal junctions cannot be the sources of these transitions.

PT2 The probability value of a transition must be between 0 and 1.

We introduce PT1 because RoboChart implements an alternating model between nondeterministic choice and probabilistic choice. (This is in line with other authors to simplify the semantic model, and in particular the interaction between nondeterministic choice and probabilistic choice—see references [18,41]. This follows, for example, the reactive modules formalism of PRISM.) Therefore, it would be awkward at best to have arbitrary probabilistic transitions in state machines. The probabilistic junctions provide a syntactic mechanism to isolate the probabilistic choices. Probabilistic choice is only made and resolved at probabilistic junctions.

3.3.2 Ⓟ Probabilistic junctions

Probabilistic junctions are also junctions, but with extra well-formedness conditions.

PJ1 There must be a probability value on every outgoing transition from a probabilistic junction.

PJ2 There must not be a guard on an outgoing transition from a probabilistic junction.

PJ3 The probability values of all outgoing transitions from a probabilistic junction must sum to 1.

PJ2 enforces construction of simple models by separating concerns. If a guard is needed, then it can be added with a preceding transition, so there is no loss of expressiveness. PJ2 is also helpful in simplifying the process of counting transitions and choice resolution; these counts are used in the notion of time in the Markov semantics and probabilistic temporal property language.

Before defining our model transformation, the next section describes the target language, the notation adopted by PRISM, its semantics, and its metamodel.

4 The PRISM notation

In the previous section, we have described how to use RoboChart to model discrete-time probabilistic systems. Another question follows: how do we verify these systems?

As mentioned, this work enables model checking using probabilistic model checkers, specifically PRISM [3], but potentially also Storm [26] and MRMC [60], which adopt the same input notation.

Section 4.1 describes the notation that supports. We then present a metamodel that we have defined for that notation in Sect. 4.2.

4.1 PRISM

PRISM[4] is a model checker for verifying probabilistic behaviour. It allows us to analyse various probabilistic models, including DTMCs and MDPs.

The PRISM language is similar to Alur's Reactive Modules formalism [27], but with a different variable model. In a reactive module, variables fall into three groups: *private*, *interface*, and *external*. A module M can read every other modules' *interface* and *external* variables, but not their *private* variables. M can write to its own *private* and *interface* variables, but not to other modules' *external* variables. PRISM does not have *private* variables: all local variables in a module are *interface* variables. PRISM also has *global* variables shared by all modules for reading or writing. Interleaving global-variable updates avoids race conditions.

PRISM employs a closed-world assumption, that is, systems are not subjected to inputs from the environment. In order to take specific environmental inputs into account, another module can be added to the model to encode generation of desired inputs.

PRISM's semantics is not compositional: parallel modules are flattened into a single system module [61]. Module combinators are similar to CSP process algebraic operators, including parallel composition, action (event) hiding, and action renaming. The parallel composition operators are as follows:

- $M_1 [\![A]\!] M_2$, requiring both modules to synchronise on the actions in the set A.
- $M_1 \parallel M_2$, which is equal to $M_1 [\![A_1 \cap A_2]\!] M_2$, where A_1 and A_2 are the sets of actions used in the modules M_1 and M_2, respectively.
- $M_1 \mid\mid\mid M_2$, which is equal to $M_1 [\![\emptyset]\!] M_2$.

Communication between modules in PRISM, however, is not based on actions. These have only names and cannot carry messages (or data). So actions are used only for synchronisation. Exchange of messages is through variables: both global and local variables.

```
 1  // S1: model type: dtmc, mdp, ctmc, ...
 2  dtmc
 3
 4  // S2: constant declarations
 5  const double pi = 3.1415926;
 6
 7  // S3: global variable declarations
 8  global i : int init 0;
 9  global j : [0..10] init 0;
10
11  // S4: formulas
12  formula f1 = pi * 2;
13
14  // S5: module definitions
15  module M1
16    // S5_1: local variable declarations
17    l : bool init false;
18
19    // S5_2: commands
20    [e1] g1 -> p1:u1 + ... + (1-p1-...):un;
21  endmodule
22
23  module M2
24  ...
25  endmodule
26
27  module Mn
28  ...
29  endmodule
30
31  // S6: rewards
32  rewards "name"
33         g1 : exp1;
34    [e1] g2 : exp2;
35  endrewards
```

Fig. 10 Structure of a PRISM model

An important feature of PRISM is statistical model checking (SMC) [62,63]. This is a Monte Carlo discrete-event simulation technique based on randomised sampling of simulations of a PRISM model. It approximates numerical and symbolic results for property checking that economises on computation. Although approximate, SMC is effective in dealing with state-space explosion. (For the robotics domain, it is also an important technique for *design-space exploration*.)

To support SMC, however, the PRISM model cannot have multiple initial states and process algebraic operators in the construction of the system module. So no parallel composition can be specified. PRISM implicitly uses $\parallel$ to define a system $M_1 \parallel M_2 \parallel \cdots \parallel M_n$ from all modules M_1, M_2, ..., M_n in the model.

The structure of a PRISM model is illustrated in Fig. 10. There are six sections in a PRISM model:

S1 is mandatory and gives the model type. In Fig. 10, S1 is given on line #2, which defines DTMC as the type for this example. Other available model types are MDP, continuous-time Markov chain (CTMC), and probabilistic timed automata (PTA).

[4] www.prismmodelchecker.org/.

S2 declares constants used in the model. In Fig. 10, S2 is given on line #5, which declares one constant pi that is of type double and set to a real number.

S3 declares global variables. In Fig. 10, S3 is given on lines #8-#9, which declare an integer variable i and a variable j with range [0..10]. Both initialised to 0.

S4 defines formulas that give names to expressions. In Fig. 10, S4 is given on line #12, which defines a formula f1 associated with expression pi*2.

S5 defines the individual modules. Each module has two subsections: S5_1 for local variable declarations and S5_2 for commands described in the sequel. S5 is given on lines #15-#29, which indicate the definition of modules M1, M2, and Mn. S5_1 is on line #17, which declares a local boolean variable l initialised to false, and S5_2 has a single command given on line #20. The command has an action e1, a guard condition g1, and a set of updates separated by +. An update is composed of an optional probability value, such as p1 in the example, and an assignment, such as u1.

S6 defines rewards associated with models. In Fig. 10, S6 is given on lines #32-#35, which define two rewards: a reward on line #33 associated with states, and another on #34 associated with transitions. Rewards, which are used in properties (but defined within models) to allow reasoning about a wide range of quantitative measures, assign particular values to certain states or transitions. If a reward has an action, then this applies to all transitions labelled by that action.

A *state* is a valuation of all variables (global and local) in the model. The *state space* S of the model is all valuations of variables. A command

```
[e] g -> p1:u1 + ... + pn:un;
```

defines a transition augmented with probabilities. The optional action e is available for synchronisation with other modules. The guard condition g ranges over the state space S. In other words, g characterises a subset of the state space, denoted by S_g, that satisfies the condition. The collection of state updates $pi : ui$, where $\sum_{1 \leq i \leq n} pi = 1$, comprises a probability pi and a collection of assignments ui. Each pair defines the probability pi of the transition going from a source state s in S_g to a target state s' specified by the set of assignments ui. A command cannot update a global variable if it has an action. This avoids two commands synchronising with different updates for a global variable.

Example 1 We present below the PRISM model for a simple robot that moves along a corridor following a random walk. It randomly chooses its moving direction: either to the left or to the right with probability p or $(1 - p)$ respectively. If, however, the robot reaches an end of the corridor, it can only move to one direction: turn around and move away from the end. Initially, the robot is at the center of the corridor.

```
1  dtmc
2
3  const double p;
4  const int N;
5
6  module random_walker
7      x : [-N..N] init 0;
8
9      [move] x = -N -> (x' = x + 1);
10     [move] x = N -> (x' = x - 1);
11     [move] (x > -N & x < N) ->
12                p : (x' = x - 1) +
13            (1-p): (x' = x + 1);
14 endmodule
15
16 rewards
17     [move] (x = N/2) : 1;
18     [move] (x = -N/2) : 1;
19 endrewards
```

Fig. 11 The PRISM model of a simple robot

The PRISM script, as shown in Fig. 11, defines a DTMC model (line #1). A constant p represents the probability described above (line #3). A constant N records half of the length of the corridor (line #4). The only module random_walker (lines #6-#14) has one local variable: x for the current location of the robot, defined over a range from -N to N, and initialised to 0 for the centre of the corridor (line #7). The commands on lines #9-#13 define the transition system of the module. In the commands, we use x′ to represent the after-transition value of x. Every move event represents one step of the robot. In the left end, the robot can move only to the right (line #9), while in the right end, it can move only to the left (line #10). In locations other than the ends ((x > -N & x < N) on line #11, where & denotes conjunction), it can move to the left (line #12) or to the right (line #13) with probability p and (1-p).

The rewards on lines #17-#18 assign a reward of 1 to every state in which the robot reaches the location N/2 (line #17) or the location -N/2 (line #18). This enables us to specify reward-based properties [64] (the properties of DTMCs or MDPs augmented with rewards) that allow quantitative measures of the model at the two locations. For example, the quantitative property R=? [F (x=0 & (X (F x=0)))], where R=? denotes reward-based operator [64], and F and X stand for the eventually and next operators in PCTL, specifies a property of the expected cumulated reward that the robot receives when it passes through the two locations before it returns to the centre of the corridor.

Next, we describe the PRISM metamodel.

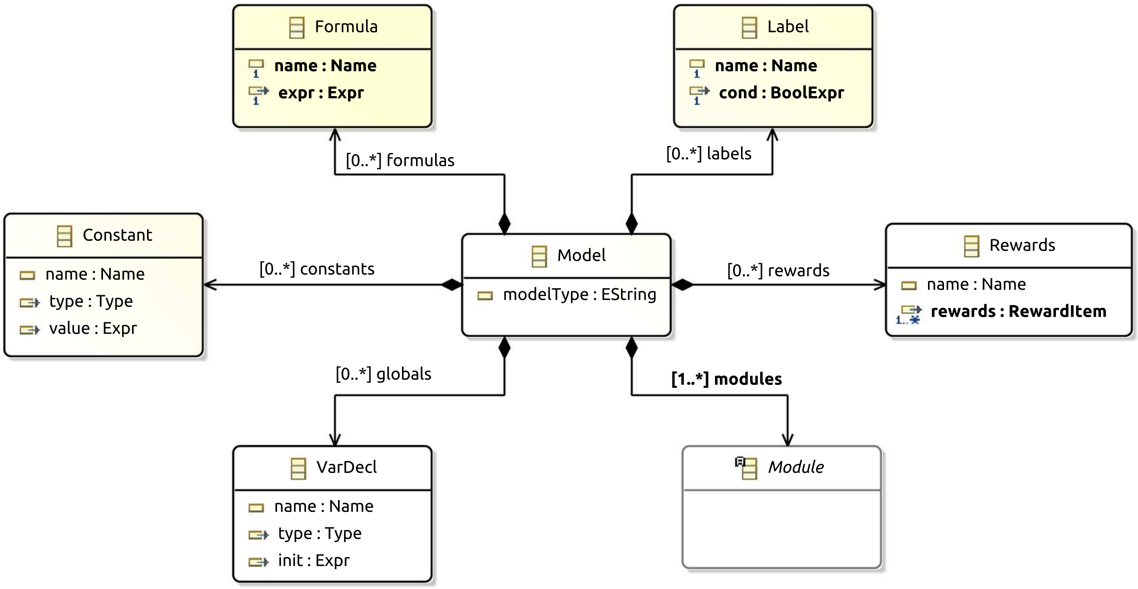


Fig. 12 Metamodel of a PRISM model

Fig. 13 Metamodel of a PRISM module

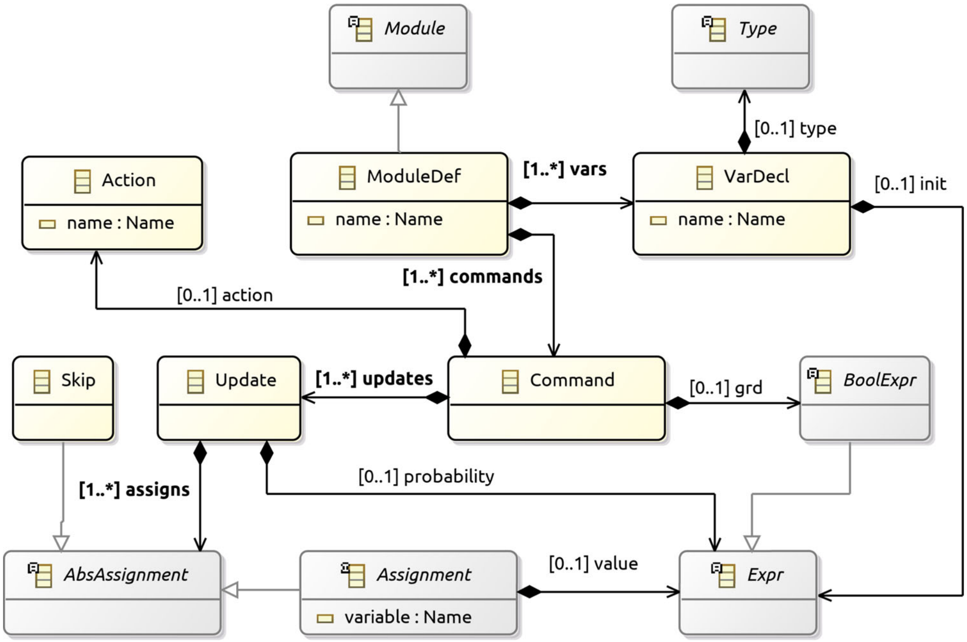
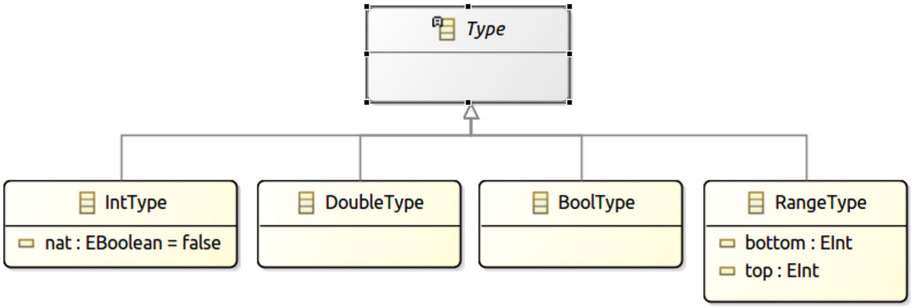


Fig. 14 Metamodel of PRISM types



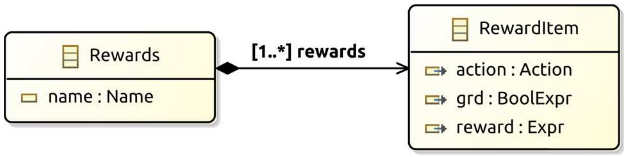


Fig. 15 Metamodel of PRISM rewards

4.2 PRISM metamodel

We show the metamodel we have defined for the structure of a PRISM model in Fig. 12. A Model contains one or more modules (Module), and zero or more constants (Constant), global variables (VarDecl), formulas (Formula), labels (Label), and rewards (Rewards).

A Constant associates a name (Name) with a type (Type) and an optional value (Expr). A VarDecl declares a variable with a name, a type, and an initial value (Expr). Formula and Label are similar but do not have a type. Both associate a (formula or label) name with an expression. A Label requires a boolean expression (BoolExpr), but a Formula can have any expression (Expr). The complete metamodel is in [1].

Figure 13 depicts the metamodel for PRISM modules. There are two ways to specify a module: either with a ModuleDefinition; or by renaming (ModuleRename, omitted in the diagram for simplicity). ModuleDef defines a module by its name, local variables (VarDecl), and commands (Command).

A command contains an optional action (Action), an optional guard (BoolExpr), and one or more updates (Update). An update has an optional probability (Expr) and a collection of assignments (AbsAssignment). There are two types of assignments: Assignment associates a variable with a value (Expr); and Skip is an empty assignment, in which no variable is updated.

There are four primitive Types, as shown in Fig. 14: boolean (BoolType), integer (IntType), range (RangeType), and double (DoubleType).

Figure 15 displays the reward metamodel. An action is optional. Without an action, the reward item assigns a reward (Expr) for all states satisfying the guard condition (BoolExpr). If there is no trigger event for these transitions, we assign the silent action "tau".

The next section defines our model transformation.

5 Model transformation

RoboChart is different from PRISM in various aspects. To deal with these differences, we structure the translation in two steps: normalisation of the RoboChart model and transformation of normalised RoboChart models to PRISM. Section 5.2 gives an overview of the translation. In Sect. 5.2.1, we define our normal form, namely restrictions on RoboChart models that ensure all transitions between two states are probabilistic, and define a strategy for normalisation. We describe a strategy to transform from a RoboChart model in normal form to a PRISM model in Sect. 5.2.2. We present formal rules used to normalise RoboChart models in Sect. 5.3 and to transform normalised models in Sect. 5.4. Extra rules are presented in “Appendix A”. The complete set of rules can be found in [1]. Not all features of RoboChart are currently supported. Section 5.1 presents our assumptions.

5.1 Translation requirements

We list the conditions that need to be satisfied by a RoboChart model for our technique to be applicable.

In DTMCs and MDPs, transitions occur in discrete-time steps. Every transition takes one unit of time. Time primitives [2] in RoboChart, however, capture budgets and deadlines using clocks and constructs like wait(n), which defines a waiting period of n units of time, and read?x<{2}, in which reading a value x through an event read is given a deadline of 2 units of time. DTMCs and MDPs do not intrinsically support clocks, time budgets and deadlines. Our first translation requirement, TR-TP1 below, therefore, rules out translation of time constructs for now.

TR-TP1 Time primitives are not used.

In addition to DTMCs and MDPs, the PRISM notation also supports probabilistic timed automata (PTAs) [65,66], which extend MDPs with the ability to model real-time behaviour through real-valued clocks [67]. The timed semantics of RoboChart, however, are based on time units [2], instead of real-valued time. The default verification method (quantitative abstraction refinement [68]) for PTAs in PRISM, therefore, cannot be used. Instead, we can use the digital clocks [69] method which uses an *integral* time model. We will extend our transformation to support PTAs to deal with time primitives in RoboChart, which is part of our future work, as discussed in Sect. 8.

TR-CN1 Connections between controllers are not asynchronous.
TR-OP1 Operations cannot be defined in controllers.
TR-ST1 States cannot have during actions.

To cater for these constructs, the PRISM model needs to include extra modules to deal with shared variables, buffers for asynchronous communication, operation calls, and interruptions of during actions by outgoing transitions. Dealing with these extra constructs is part of our agenda for future work.

TR-TY1 Only primitive types and enumerations, sequences of these types, or sequences of sequences of these types are used.

TR-EX1 Quantification, lambda, and definite description expressions cannot be used since the PRISM notation is concrete.

PRISM supports only integer, boolean, and real numbers. Refinement techniques [70,71] are a possible solution to deal with abstract data types and constructs in RoboChart. For example, we can refine a variable of the type $\mathbb{P}\,T$, where T is an enumeration, in RoboChart to a variable over an integer range in PRISM, and use it as a bitmap. Support of more abstract data types and expressions is part of our plans for future work.

In the next section, we give an overview of our translation strategy, which is formalised later in Sects. 5.3 and 5.4. We deal with a rich set of features: parallel controllers, with parallel hierarchical machines, nondeterminism, junctions, probabilistic junctions, synchronous connections, input and output triggers, and functions.

5.2 Overview

In this section, we first present the normal form that we define for RoboChart models and the strategy for normalisation in Sect. 5.2.1, and then we describe the strategy for their transformation to PRISM models in Sect. 5.2.2.

5.2.1 Normalisation

A normalised RoboChart model satisfies the following extra conditions on transitions and junctions.

NFM-1 A state has at least one outgoing transition.

NFM-2 A transition that is from a state or a normal junction to a normal junction has an action.

NFM-3 A transition can have a trigger or an action, but not both together.

NFM-1 is needed due to the fact that DTMCs and MDPs are stochastic. Any state in a PRISM model has to have at least one outgoing transition.

If an incoming transition to a normal junction has no actions, this transition can be combined with outgoing transitions of the junction, which can be removed. The combination of transitions and the removal of junctions reduce the translated PRISM models.

For transitions with a trigger and an action, the action may a) update global variables, and b) have input and output events, or synchronisations, such as the action in the label close[g]/collect. Such transition cannot be translated to a PRISM command directly because a command a) cannot have an action and an update to global variables together, and b) cannot have more than one action. So we impose NFM-3.

To illustrate normalisation, we use the taskSTM machine in Fig. 4 and the movingSTM machine in Fig. 6 as examples. We present their normal forms, and describe how to get a normalised machine.

The taskSTM machine is not in normal form because the transition with a trigger request?g has an action as well, which does not satisfy the condition NFM-3.

A normalised version of taskSTM is shown in Fig. 16 where the changed parts are highlighted in a dashed box. The original transition is replaced by a transition to a new probabilistic junction, and a transition from the probabilistic junction to the original target. The condition of the original transition is used in the transition to the new probabilistic junction, ensuring that a transition is not taken unless the guard holds. The single transition from the probabilistic junction has probability 1 and the action of the original transition. So this new transition is always taken, and the action is executed, as in the original model.

The movingSTM machine has an initial junction, a normal junction, and two states: a simple state Stuck and a composite state Move. The machine movingSTM is not in normal form because a) the incoming transition of the junction in the machine has no action, which does not satisfy the condition NFM-2, and b) Stuck has no outgoing transitions, which does not satisfy NFM-1.

A normalised version of movingSTM is shown in Fig. 17 where the changes are highlighted in a dashed box. The junction and its corresponding incoming and outgoing transitions in the original machine are replaced by two transitions: one from Move to Stuck, and another from Move to itself. They are guarded by the conjunction of the guards of the transitions to and from the junction. The target states are the same.

In the normalised machine, the Stuck state has a new outgoing transition to a new state loop. In general, we add a state loop for each state machine or composite state that has at least one state with no outgoing transition. The new transition (from Stuck in our example) has no trigger, guard, or action. So this transition introduces no new visible behaviour. From loop, a similar transition leads back to loop itself. Overall, where the original machine deadlocked, the normalised machine livelocks. As already mentioned, this is what is required for a probabilistic analysis using DTMCs and MDPs. In the example, there is no loop state in Move because all its substates have outgoing transitions.

In Sect. 5.3, we present the rule that can be used to normalise the original model if applied exhaustively (to all its state machines). Before presenting that rule, however, we give an overview of the normalisation process, explaining how each of the transitions and junctions of movingSTM are affected by it.

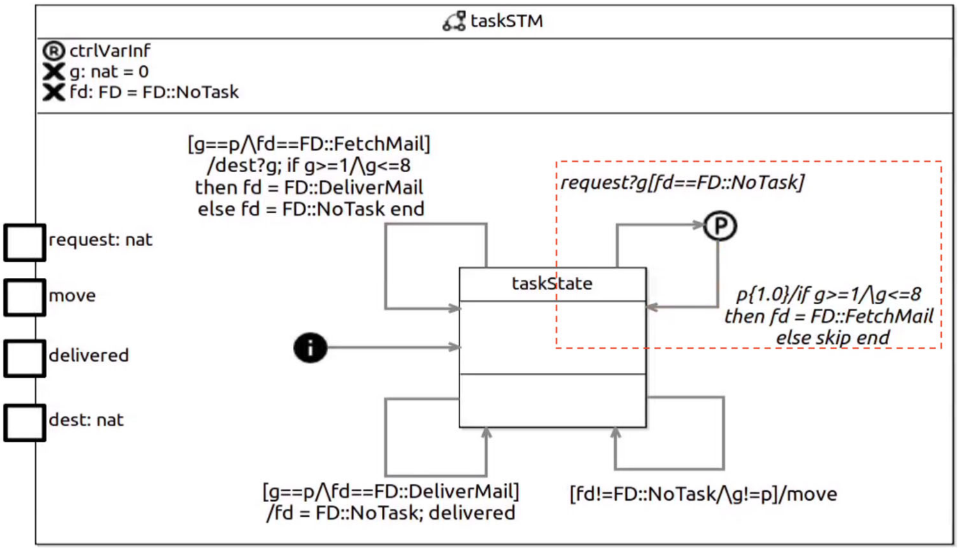


Fig. 16 Normal form of task state machine (the changed parts are highlighted in a dashed box)

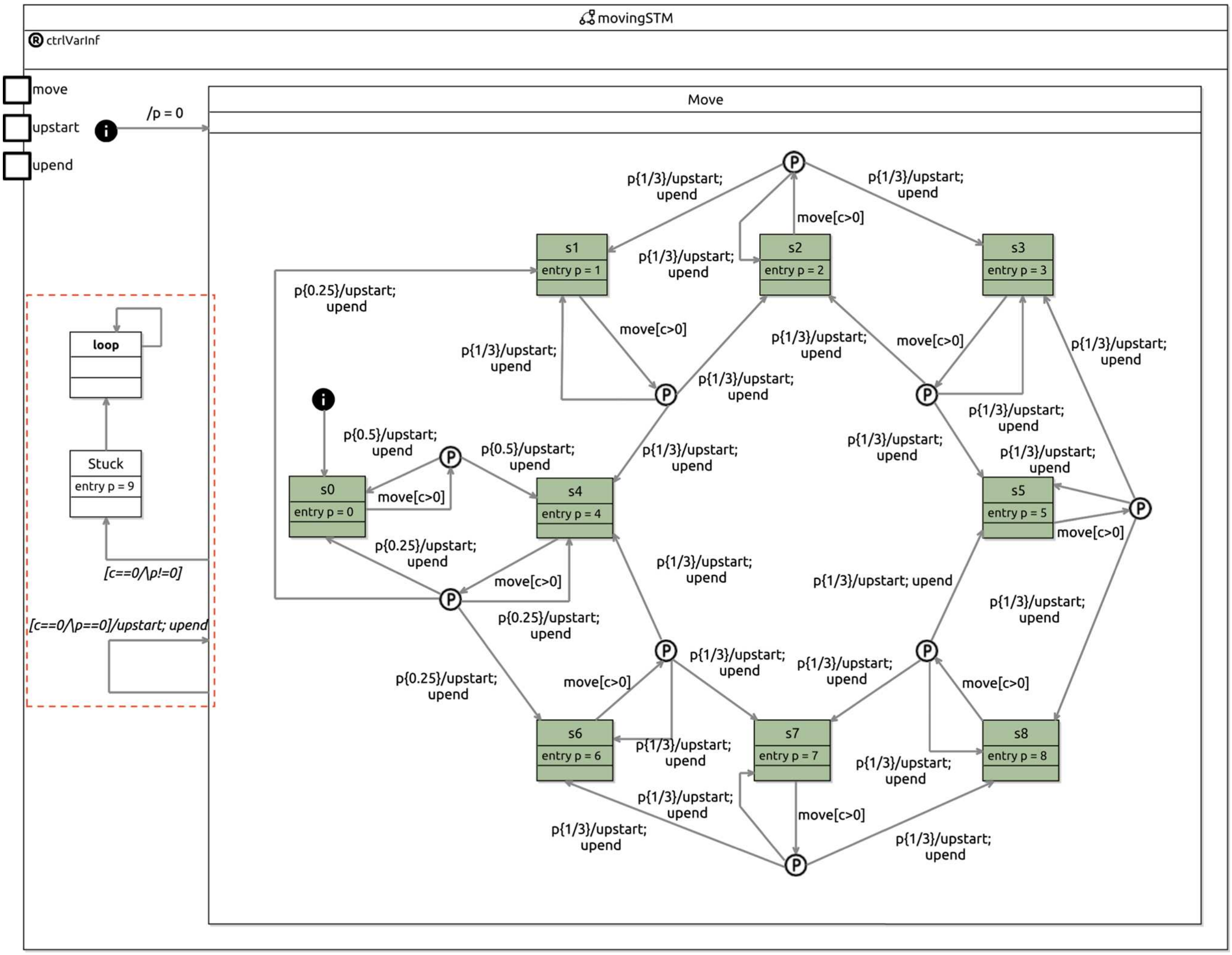


Fig. 17 Normal form of movement state machine (the changed parts are highlighted in a dashed box)

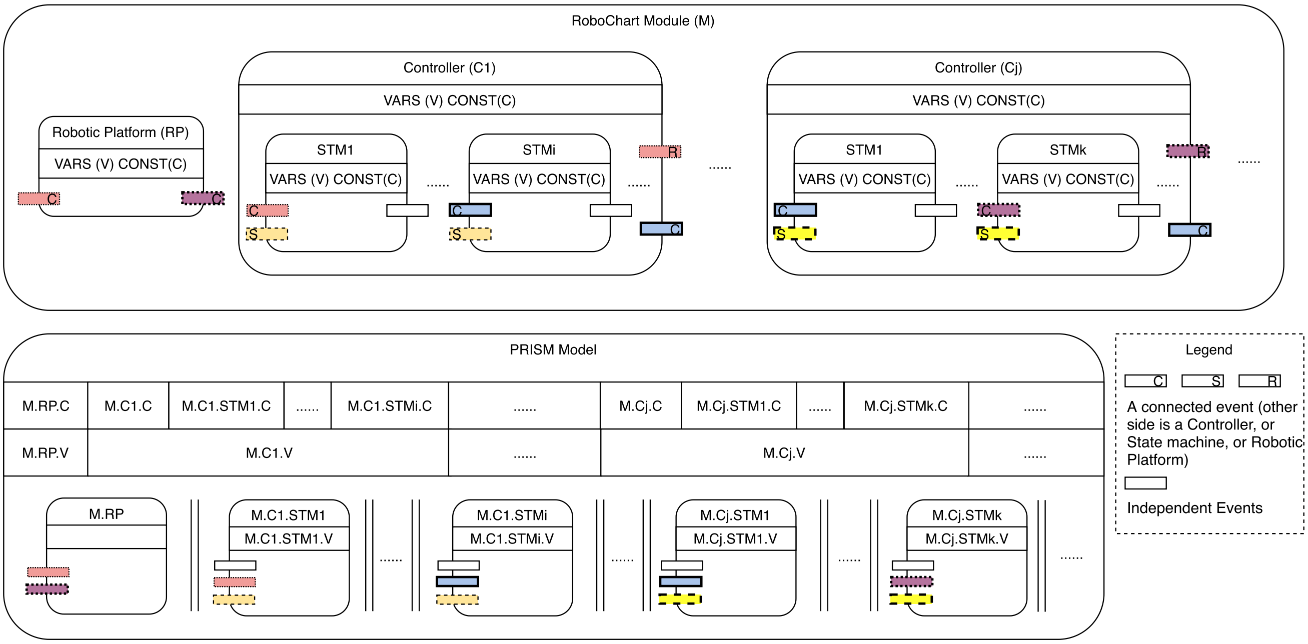


Fig. 18 Structure of translation (V for variables, C for constants, and small boxes for events on either sides of Robotic Platform, Controllers, or State machines)

Each state machine is normalised by a) adding the loop state and corresponding transitions, if needed; b) combining incoming and outgoing transitions of normal junctions as illustrated for the transition out of Move in movingSTM; c) splitting each transition that has a trigger and an action together into two transitions connected via a new probabilistic junction as illustrated for a self transition of taskState in taskSTM; and d) normalising machines of composite states.

As illustrated above, the combination of transitions to satisfy NFM-2 may lead to the introduction of multiple outgoing transitions from the original source junction with stronger guards. Because there can be only one outgoing transition from an initial junction, we cannot combine outgoing transitions of initial junctions in this way. For probabilistic junctions, PJ2 forbids the presence of guards on outgoing transitions. We cannot, therefore, consider the optimisation enabled by NFM-2 for transitions from initial or probabilistic junctions.

The use of normalisation deals with all the complexity of a RoboChart model to produce a module in a form that is convenient for a direct translation to PRISM.

5.2.2 Transformation to PRISM

The structure of our translation is sketched in Fig. 18. The structure of a RoboChart model is illustrated at the top, and the structure of its corresponding PRISM model is given at the bottom on the left.

The RoboChart module, M, contains a robotic platform RP and multiple controllers (C1, …, Cj, …). Each controller contains various state machines (STM1, …, STMi or STMk, …). The robotic platform, controllers, and state machines all have a declaration part, which includes variables, constants, and events. We distinguish events in four groups, we use: a) a blank box for independent events that are not connected; b) a box with a C inside for the events that are connected from or to a controller; c) a box with an R inside for the events that are connected from or to the robotic platform; and d) a box with an S inside for the events that are connected from or to a state machine. Additionally, boxes drawn with the same sort of lines (solid, dashed, and so on) indicate events that are connected (either directly or indirectly). For example, the machine STM1 in C1 has a dotted box (C), a dashed box (S), and a blank box. The dotted box denotes the events of the machine that are connected to the events of its controller C1 that are represented by a dotted box too. The dotted box of C1, however, has an R, so these events are further connected to those of the robotic platform (represented by the same dotted box). The dashed box (S) of STM1 represents the events that are connected to other machines in C1, specifically STMi. Particularly, STMi of C1 contains events represented by the bold box (C) that are connected to C1, then to Cj, and finally to the events of STM1 in Cj represented also by a bold box (C).

Generally, the resulting PRISM model (see bottom left of Fig. 18) contains three parts: (a) constants that correspond to the constants of the robotic platform, denoted M.RP.C in

Table 3 Summary of construct syntax in rules

Form	Description
Junction	The class name from the metamodel also represents a collection of objects that have the Junction type. Similarly, State, ProbabilisticJunction, Controller, and so on. Particularly, ProbJunc is an abbreviation for ProbabilisticJunction for a compact space.
Class_{rc}	Subscripts indicate classes of different metamodels, where rc stands for RoboChart, and pr for PRISM. For example, Transition_{rc} and ModuleDef_{pr} denote the Transition in RoboChart, and the ModuleDef in PRISM.
Class_{pr}	
$\mathbb{P}X$	Power set of X.
$X \times Y$	Cartesian product.
$\{x : T \mid P \bullet e(x)\}$	Set comprehension.
$\{x : T \mid P\}$	Defined as $\{x : T \mid P \bullet x\}$.
$\{x : T \bullet e\}$	Defined as $\{x : T \mid true \bullet e\}$.
$(\mu x : T \mid P)$	Definite description.
$\bigcup\{x : T \bullet e(x)\}$	Generalised union, where $e(x)$ is a set expression.
$p.n$	Selection of nth element from the tuple p.
$obj.v$	Selection of value of component v from object obj.
$\langle f_1 \rightsquigarrow v_1, \cdots \rangle_C$	Record where f_i is a component name and v_i is the value associated to that component. It represents an object of the class C with its components instantiated.
$R ⦇ S ⦈$	Relational image of a relation R under a set S.
$[\![i : T]\!]_{\mathcal{X}}$	Transformation of i to PRISM in the context $\mathcal{X}$. For example, $[\![\cdots]\!]_{\mathcal{M}}$ denotes the module context.
$uname(par, m)$	This function constructs a fresh unique identifier for a new element from the supplied construct par (of type NamedElement) and the name m (a string) of the element.
$uname(par, m, n)$	Similar to $uname(par, m)$, but this function has three parameters where both m and n are of type string.
$id(n)$	This function defines a unique identifier for an existing construct n (of type NamedElement). If n is null, it gives an empty name. One possible implementation is to use qualified names.

Fig. 18, where we use qualified names for all components (see id in Table 3), the controllers (`M.C1.C`), and the state machines (`M.C1.STM1.C`) in the RoboChart model; (b) global variables that correspond to the variables of the robotic platform (`M.RP.V`) and the controllers (`M.C1.STM1.V`); and (c) a variety of modules that are in parallel (denoted as parallel lines): one (`M.RP`) corresponding to the robotic platform and others corresponding to state machines. The local variables of a state machine become local variables of the corresponding module, such as `M.C1.STM1.V` in the `M.C.STM1` module.

Renaming plays an important role in the translation of a RoboChart model to PRISM because the variable model of RoboChart is different from that of PRISM. Each element in RoboChart has a given scope. For example, a variable defined in a controller is visible to its inner state machines, but not to the state machines in other controllers. Two variables of the same name in different controllers or state machines are permitted. In contrast, the PRISM language has a flat structure. If two variables in different modules have the same name, then this results in a name conflict. To address this problem, we rename all elements, with the exception of events, used in a RoboChart model to ensure that they all have unique names before translation.

In RoboChart, connections can associate events of different names. This means that a component (machine, controller, or robotic platform) can be used in any context, irrespective of the names that they use for their events. In PRISM, however, connections are realised via synchronisation based on the names of the actions. So, we must make sure that actions in different PRISM modules that need to synchronise have the same name. In our translation, we represent RoboChart events using PRISM actions and connections using synchronisation. Therefore, we ensure, before translation, that events are renamed so that all connections associate events of the same name.

For example, the independent events (represented by the blank box) in STM1 of C1 become independent actions represented in the same style in the `M.C1.STM1` module. The actions represented by the dotted box in the module have the same names as the actions represented by the same type box in the robotic platform module. So the modules `M.C1.STM1` and M.RP synchronise on these actions, which reflects the connection of their corresponding events in the RoboChart model. Other actions represented by a dashed box, or a bold box, or a bold dashed box are handled similarly.

To illustrate the overall structure of the translation, we use the mail delivery robot model introduced in Sect. 3. We present the structure of its PRISM script, and describe how to get it from the RoboChart model.

The PRISM script is sketched in Fig. 19, where elements not relevant to the structure are omitted, such as the naming, constant declarations, and command details. The model type is on line #1; we use the type DTMC so that we can obtain probabilities of the robot running out of power at different offices. The model type is a parameter to the translation.

A translated PRISM model can also use another model type: MDP. The selection of the model type depends on the properties to be analysed using PRISM. For DTMC, PRISM randomises nondeterministic choice and regards them as uniformly probabilistic choices. So PRISM can check quantitative properties such as "what is the probability of the robot running out of power and stay stuck at office 2". For MDP models, PRISM does not treat nondeterministic choice probabilistically, and aims for establishing minimum and maximum probabilities, not exact probabilities.

Constants and global variables are declared on lines #3-#7, and four modules are defined on lines #9-#43. The module deliverRP on lines #9-#20 corresponds to the robotic platform, and the other three modules on lines #21-#27, #28-#33, and #34-#43 correspond to three state machines in the RoboChart model.

Constants and variables provided by the robotic platform are translated to constants and global variables, as shown on lines #4-#7. MININT and MAXINT are placeholders for the smallest and largest integer to be included in the set to represent the integer numbers. Their values are parameters for the translation.

The robotic platform module deliverRP in PRISM represents the environment of the system modelled, and generates inputs (using a nondeterministic choice over possible input values) to the software. Two local variables on lines #10-#11 correspond to the output events request and dest of the robotic platform (see Fig. 2), that is, the inputs to the controller. They have the same type as the request and dest events. MAXNAT is similar to MAXINT but represents the largest natural number. When a synchronisation on either action (request or dest) occurs, the corresponding local variable (either EVT__request or EVT__dest) is assigned a value nondeterministically chosen. The commands on lines #13-#15 define the choice for EVT__request. The other module involved in the synchronisation (in this case, taskSTM) copies the value to a local variable (as illustrated by the command on line #39) to simulate the acceptance of input values from the environment.

```
 1  dtmc
 2
 3  const int NoTask = 0;
 4  const int batteryCapacity;
 5  const int chargeStep;
 6  global c : [MININT..MAXINT];
 7  global p : [MININT..MAXINT];
 8  ...
 9  module deliverRP
10      EVT__request : [0..MAXNAT];
11      EVT__dest : [0..MAXNAT];
12      ...
13      [request] true -> EVT__request' = 0;
14      [request] true -> EVT__request' = 1;
15      [request] true -> EVT__request' = 2;
16      ...
17      [dest] ...
18      [delivered] true -> true;
19      ...
20  endmodule
21  module movingSTM
22      ...
23      [move] ...
24      [upstart] ...
25      [upend] ...
26      ...
27  endmodule
28  module batterySTM
29      ...
30      [upstart] ...
31      [upend] ...
32      ...
33  endmodule
34  module taskSTM
35      g : [0..MAXNAT] init 0;
36      fd : [0..2] init NoTask;
37      ...
38      [move] ...
39      [request] ... -> (g'=EVT__request)& ...;
40      [delivered] ...
41      [dest] ...
42      ...
43  endmodule
```

Fig. 19 Structure of the mail delivery robot PRISM script

The actions move, upstart, upend, request, dest, and delivered in four modules correspond to the events in the RoboChart model. Since in this example only events of the same name are connected, there is no need to rename events. The actions of the same name such as move on line #23 in movingSTM and line #38 in taskSTM allow synchronisation of the two modules, as specified by the connection in Fig. 3.

The events upstart and upend are used for synchronisation between two state machines. They are not events of the robotic platform; therefore, the visible behaviour of the RoboChart module does not include occurrences of these events. In PRISM, it is possible to use a hiding operator (similar to that of CSP) to make synchronisations on given actions internal (not visible). The use of such operator, however, makes statistic model checking infeasible. So, we keep these actions visible.

Two local variables (g and fd) of the taskSTM module on lines #35-#36 are translated from the corresponding local variables in taskSTM (see Fig. 4). The type of fd is a range that relates to the number of literals in the enumeration FD (see Fig. 4). The initial value of fd is NoTask that is represented by 0 (line #3).

We illustrate our approach to modelling state machines in PRISM in Fig. 20, where we present the module for the normalised taskSTM machine in Fig. 16 on lines #25-#54. We use three (sets of) variables: program counter scpc, lock, and exit variables, in addition to the local variables of the state machine. In this simple example, we have local variables fd and g, and just two variables scpc and lock. Later in this section, we consider examples with extra variables including exit variables. The required variables c and p, and the required constants batteryCapacity and chargeStep are declared globally, as shown in Fig. 19.

A program counter variable scpc records the current state of the machine. In our example, we have just the initial state and the taskState. We name the initial state i0 and declare two constants i0 and taskState to associate these states with the numbers 1 (line #12) and 2 (line #13). Extra constants declared on lines #14-#23, associated with the numbers from 3 to 12, are needed because the notion of state in RoboChart does not match that of PRISM (see Sect. 4.1), which is based on valuation of variables. So, we have constants for each action of RoboChart whose encoding in PRISM may lead to a change of the value of the variables in the scope of the machine module taskSTM.

We associate these constants with a sequence of integer numbers starting from 0 and increasing in increments of one. For convenience, we refer to these numbers as *state numbers* because they are used to index states in machine modules. In principle, all that is required is that the constants uniquely identify a module state. The definition of the type of the program counter variable, scpc, which needs to range over the values of these constants, however, imposes an extra challenge. In PRISM, the type of a variable must be finite, which means integer numbers (int) cannot be the type of scpc. MININT and MAXINT, previously described, are used to define the set of integer numbers in RoboChart models. The state numbers, however, might be larger than MAXINT. We, therefore, number the states sequentially, and use specifically a contiguous range of integers, starting at 0, such as [0..12] shown on line #26, to specify all values that scpc can take. The upper limit 12 of the interval is the maximum number associated with these state constants in the module. Any finite set would be appropriate for the translation, but that set needs to be identified to define the constants and the type of scpc. Our particular approach is based on an integer interval starting at 0.

In principle, the execution of the commands for each transition of the machine can lead to a change of state of the PRISM module, because a transition can have an action. So, each transition is given a set of numbers used to record the states of the PRISM module that are reached when the encoding of the transition is executed. Figure 21 presents an annotated version of the normalised taskSTM where the transitions are associated with the numbers used in the PRISM module in Fig. 20 to encode the transitions.

For example, the transition labelled t1 in Fig. 21 is associated with the state 2, for taskState, and also, with an extra state 6, due to the action move. The transition t2 is associated with states 9, 10, 11, and 12; similarly, t3 and t4 are associated with other states. The names of the constants representing the extra states are based on the name of the source state of the transition. In the example, we have taskState_3, taskState_6, and so on. When the source is the probabilistic junction, we use p0_1 and p0_2. What is essential is a mechanism to give unique names to the extra-states constants.

Since the execution of a RoboChart transition may take the PRISM module through several states, we need a mechanism to prevent the interference from other transitions when the machine module is in one of these intermediary states. For instance, if t4 is taken, and the module is in the state 4, other transitions should not be taken until t4 is completed by returning to taskState. For this reason, we introduce a lock variable (line #27) for the machine module. This variable initially takes the value 0, represented by a constant LOCK_FREE (declared on line #5). It records that there is no transition in execution, and, therefore, other transitions can be taken. In addition to 0, lock can also take other values in the range 1 to 5 (line #6-#10). Each number represents a transition. For example, 5 (represented by T4 on line #10) denotes the transition t4. For convenience, we call these numbers *transition numbers* because they are used to index transitions in machine modules. The need to record in lock a transition number is not illustrated in this example, but it is explained later when the example for movingSTM is presented.

In the initial state of the PRISM module, the values of the local variables scpc and lock are i0 and LOCK_FREE. The PRISM transition system simulates the behaviour of the machine. In the initial state, the single transition t0 is available and can be taken. So, when the value of scpc is i0, it can be changed to taskState as shown by the command on line #31.

When scpc records that the current state is taskState, there are four transitions available, captured by the commands on lines #33, #36, #43, and #47. They all have guard conditions to ensure scpc is taskState and lock is free. Other guard conditions in these commands are from the guards of the corresponding transitions. Particularly, the

```
1  const int NoTask = 0;
2  const int FetchMail = 1;
3  const int DeliverMail = 2;
4
5  const int LOCK_FREE = 0;
6  const int T0 = 1;
7  const int T1 = 2;
8  const int T2 = 3;
9  const int T3 = 4;
10 const int T4 = 5;
11
12 const int i0 = 1;
13 const int taskState = 2;
14 const int p0 = 3;
15 const int p0_1 = 4;
16 const int p0_2 = 5;
17 const int taskState_3 = 6;
18 const int taskState_4 = 7;
19 const int taskState_5 = 8;
20 const int taskState_6 = 9;
21 const int taskState_7 = 10;
22 const int taskState_8 = 11;
23 const int taskState_9 = 12;
24
25 module taskSTM
26   scpc : [0..12] init i0;
27   lock : [0..5] init LOCK_FREE;
28   fd : [0..2] init NoTask;
29   g : [0..MAXNAT] init 0;
30   // The transition [t0] from [i0] to [taskState].
31   [] (scpc=i0) -> (scpc'=taskState);
32   // The transition [t1] from [taskState] to [taskState].
33   [] (scpc=taskState)& (lock=LOCK_FREE)& (fd!=NoTask)& (g!=p) -> (lock'=T1)& (scpc'=taskState_3);
34   [move] (scpc=taskState_3) -> (scpc'=taskState)& (lock'=LOCK_FREE);
35   // The transition [t2] from [taskState] to [taskState].
36   [] (scpc=taskState)& (lock=LOCK_FREE)& (g=p)& (fd=FetchMail) -> (lock'=T2)& (scpc'=taskState_6);
37   [dest] (scpc=taskState_6) -> (g'=EVT__dest)& (scpc'=taskState_7);
38   [] ((scpc=taskState_7))& ((g>=1)& (g<=8)) -> (scpc'=taskState_8);
39   [] (scpc=taskState_8) -> (fd'=DeliverMail)& (scpc'=taskState)& (lock'=LOCK_FREE);
40   [] ((scpc=taskState_7))& (! (g>=1)& (g<=8)) -> (scpc'=taskState_9);
41   [] (scpc=taskState_9) -> (fd'=NoTask)& (scpc'=taskState)& (lock'=LOCK_FREE);
42   // The transition [t3] from [taskState] to [taskState].
43   [] (scpc=taskState)& (lock=LOCK_FREE)& (g=p)& (fd=DeliverMail) -> (lock'=T3)& (scpc'=taskState_4);
44   [] (scpc=taskState_4) -> (fd'=NoTask)& (scpc'=taskState_5);
45   [delivered] (scpc=taskState_5) -> (scpc'=taskState)& (lock'=LOCK_FREE);
46   // The transition [t4] from [taskState] to [p0].
47   [request] (scpc=taskState)& (lock=LOCK_FREE)& (fd=NoTask) ->
48         (g'=EVT__request)& (lock'=T4)& (scpc'=p0);
49   // From the probabilistic junction [p0]
50   [] (scpc=p0) -> (scpc'=p0_1);
51   [] ((scpc=p0_1))& ((g>=1)& (g<=8)) -> (scpc'=p0_2);
52   [] (scpc=p0_2) -> (fd'=FetchMail)& (scpc'=taskState)& (lock'=LOCK_FREE);
53   [] ((scpc=p0_1))& (!(g>=1)& (g<=8)) -> (scpc'=taskState)& (lock'=LOCK_FREE);
54 endmodule
```

Fig. 20 The PRISM module of taskSTM

transition t4 has a trigger request?g, which is recorded in the command on line #47 with an action `request` and an update to the local variable `g` from the value of the variable `EVT__request` to simulate a communication between the machine and the robotic platform. If these transitions are enabled and taken, the updates of these commands change the value of `lock` to the corresponding transition number and the value of `scpc` to the first state number corresponding to the transition. For example, the command on line #33 updates `lock` to `T1` and `scpc` to 6 (`taskState_3`). After one of these commands is executed, the value of `lock` is not `LOCK_FREE` any more, therefore, none of these commands

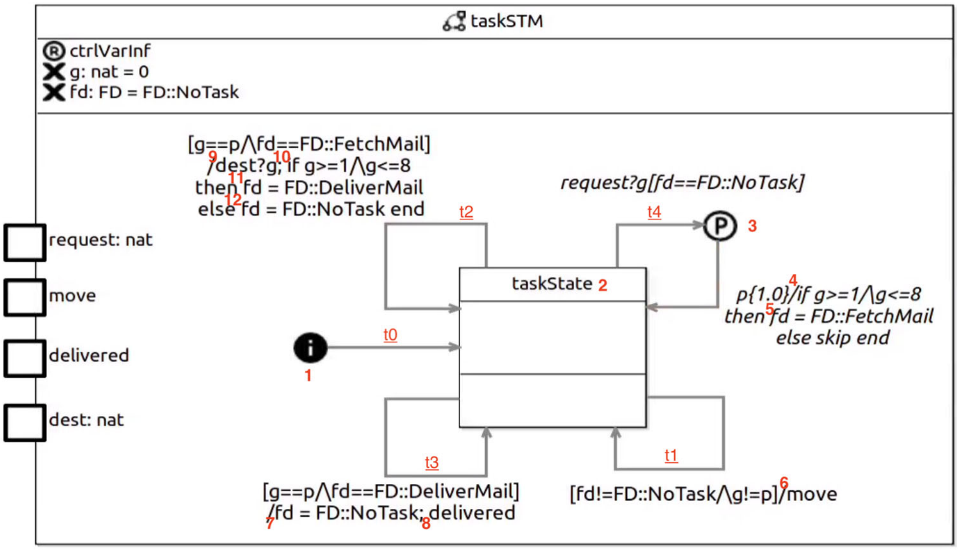


Fig. 21 Annotated normal form of task state machine

are no longer enabled to avoid interrupting the taken transition.

Each transition corresponds to several commands in the PRISM module. The transition t1 is encoded by two commands on lines #33 and #34. The first command deals with scpc and lock as just explained. When the current state of the machine is taskState_3, only the second command on line #34 is enabled. The action move of the command corresponds to the action move of the transition. After the synchronisation of the move action, the command changes the values of scpc and lock to taskState and LOCK_FREE to encode that the transition is completed and its target state is entered.

The transition t2 is implemented by six commands on lines #36-#41. After it is taken, scpc and lock are updated to taskState_6 and T2 on line #36. As shown in Fig. 21, a sequential composition of actions is encountered at taskState_6. The first action dest?g is encoded by the command on line #37, while the second conditional action corresponds to the subsequent commands on lines #38-#41, of which two are for the if branch and two for the else branch. In each pair, the first command is related to the branch condition and the second to its action. When the module is at the state taskState_7 because of the synchronisation on dest, if the value of g is valid (between 1 and 8), the command on line #38 is executed next, which takes the module to taskState_8. At this state, only one command on line #39 is enabled. Upon its execution, fd becomes DeliverMail. If the value of g is not valid, fd is set back to idle (NoTask) on lines #40 and #41.

The transition t3 is enabled if the destination of the delivery is reached (g is equal to p) and the robot is delivering as shown by the command on line #43. After the transition is taken, the module passes through states taskState_4 and taskState_5. Finally, the task returns to NoTask and a delivered event is signalled, shown by the commands on lines #44 and #45.

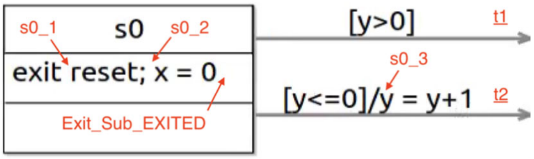


Fig. 22 A simple state with an exit action

The transition t4 has an input trigger (request?g) and its target is a probabilistic junction represented by the constant p0 on line #14. The command on lines #47 and #48 corresponds to t4; it updates scpc to p0. The probabilistic junction has one outgoing transition to taskState with probability 1. The action of this transition is a conditional that involves two states p0_1 and p0_2. From p0, the next state is p0_1, as shown on line #50. The conditional action is encoded by the commands on lines #51-#53. The encoding is similar to that for the transition t2, but since the action of the else branch is skip, there is no additional state, and, therefore, only one command encodes that branch.

Since none of the states in the taskSTM machine has an exit action, its PRISM module in Fig. 20 has no exit variable. If a state has an exit action, such as the state s0 in Fig. 22, an exit variable is used in PRISM, as shown in Fig. 23. Constants s0_1, s0_2, and s0_3, declared on lines #9-#11, represent three extra states in PRISM as annotated s0_1, s0_2, and s0_3 in Fig. 22. Intuitively, we can think of the exit action as being part of the action of every transition out of the state. So, we deal with exit actions much in the same way as we deal with transition actions.

```
 1  const int Exit_NONE = 0;
 2  const int Exit_Sub_ACT = 4;
 3  const int Exit_Sub_EXITED = 6;
 4  const int LOCK_FREE = 0;
 5  const int T1 = 1;
 6  const int T2 = 2;
 7  ...
 8  const int s0 = 2;
 9  const int s0_1 = 3;
10  const int s0_2 = 4;
11  const int s0_3 = 5;
12
13  module ...
14    exit : [0..6] init Exit_NONE;
15    ...
16
17    // The transition [t1] from [s0] to [...]. Step 1: trigger an exit of s0
18    [] (scpc=s0)& (lock=LOCK_FREE)& (y>0) -> (lock'=T1)& (exit'=Exit_Sub_ACT);
19    // Step 3: make sure the simple state [s0] has been exited.
20    [] (scpc=s0)& (lock=T1)& (exit=Exit_Sub_EXITED) -> (scpc'=...);
21    // The transition [t2] from [s0] to [...]. Step 1: trigger an exit of s0
22    [] (scpc=s0)& (lock=LOCK_FREE)& (y<=0) -> (lock'=T2)& (exit'=Exit_Sub_ACT);
23    // Step 3: make sure the simple state [s0] has been exited.
24    [] (scpc=s0)& (lock=T2)& (exit=Exit_Sub_EXITED) -> (scpc'=s0_3);
25    // Step 2: exit of s0
26    [] (scpc=s0)& (exit=Exit_Sub_ACT) -> (scpc'=s0_1);
27    [reset] (scpc=s0_1) -> (scpc'=s0_2);
28    [] (scpc=s0_2) -> (x'=0)& (exit'=Exit_Sub_EXITED);
29    ...
30  endmodule
```

Fig. 23 The PRISM snippet of the simple s0 in Fig. 22

The snippet in Fig. 23 encodes the exit of the transitions t1 and t2 in Fig. 22 from their source state s0. The commands on lines #18 and #20 are specific to t1, and those on lines #22 and #24 are for t2. Both transitions share an exit from s0, as encoded on lines #26-#28, using an exit variable declared on line #14. This variable records the exit states that arise by the execution of the exit action and takes three possible values in this simple example. They represent the states in which the execution of the exit action has not started (Exit_NONE), has started (Exit_Sub_ACT), and has completed (Exit_Sub_EXITED). These values are represented by three constants declared on lines #1-#3.

If t1 is taken, lock is set to T1 and an exit request is triggered by updating exit to Exit_Sub_ACT, as shown on line #18. The next command on line #20 for t1 checks the exit completion indicated when the value of exit is Exit_Sub_EXITED. The value Exit_Sub_ACT indicates an exit request to the current state of the module. Since t1 is taken, the current state is s0 (that is, scpc=s0). So the guard on line #26 is satisfied, and s0 receives the request and updates the counter to s0_1 to deal with its exit action in the commands on lines #27-#28. In the end of its exit action, s0 marks that it has exited by setting exit to Exit_Sub_EXITED (on line #28). The command on line #20, therefore, is enabled. So after exiting from its source state, t1 continues to deal with its transition action and targets, omitted in Fig. 23. The way we encode t2 is very similar.

We use this simple state s0 with an exit action to demonstrate how an exit variable is used in our PRISM models to deal with the exit of states. The extra exit variable, however, is not needed to deal with state exit. We could use the counters; in this example, we could set the value of scpc to the extra state numbers, such as s0_1, in the first command corresponding to a transition: the command on line #18 or #22, for example. The last command, such as that on line #28, then would set the counter to the state just after the exit from the source state, such as s0_3. The exit variables are, however, necessary to deal with composite states.

In the next example, in Fig. 24, we consider a transition t0 whose source state S0 is composite. The execution sequence of the transition starts with a request for S0 to exit. That state then passes the request into its inner states recursively. Upon receipt of an exit request, the innermost (simple) substate executes its exit action (if any). Afterwards, each enclosing state repeats this process up to the source state.

In Fig. 24, all states have an exit action and at least one outgoing transition. In exiting S0, the exit actions of its substates are executed sequentially: the exit action of S2, the exit action of S1, and the exit action of S0, if the current substate of S1 is S2, or the exit action of S3, the exit action of S1,

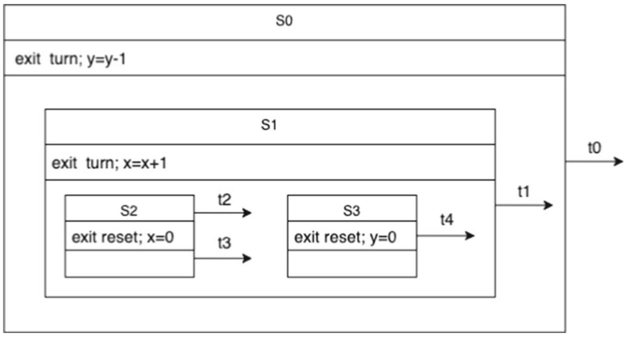


Fig. 24 An example to illustrate the need of exit variables

and the exit action of S0, if the current substate of S1 is S3. One possible approach to encode this uses sets of PRISM commands for each of the exiting paths, with corresponding guards to determine the current substates. This leads to duplication of extra states introduced by exit actions. In this example, the exit action of S1 is shared by both paths to exit. We could encode this by introducing an extra state in the beginning of the exit action, such as `S1_exiting`, and then the end of the encodings of the exit actions of S2 and S3 leads to `S1_exiting`. To use this approach, however, the encoding of the transition t0 needs to take into account the internal structure of S0. This is neither straightforward nor compositional.

Instead, we use a staged approach, in which the encoding of a transition (like t0) triggers an exit request to its source state (S0), and waits for it to exit. It is the encoding of the source state that reflects its internal structure. For this example, the commands for S0 deal with an exit request, relay the request to its current substate S1, and wait for S1 to exit. The commands for S1 are similar. The commands for S2 deal with the exit request from S1, execute its exit action, and exit S2 using the approach in Fig. 23 because S2 is simple. Afterwards, the command for S1 that waits for S2 to exit becomes enabled. After it is taken, the encodings for its exit action are executed, and so exit S1. In a similar way, the encoding for S0 controls its exit.

To control this exit flow in PRISM, we introduce an extra `exit` variable for each composite state, and define the exit of a composite state in six stages:

EX-S1 it is not in an exit flow;
EX-S2 it gets an exit request either from a transition or from its parent;
EX-S3 it passes the exit request to its substates;
EX-S4 it waits for its substates to exit;
EX-S5 its substates have exited (at this point, its exit action is executed); and
EX-S6 it is exited (after execution of its exit action).

To illustrate our approach to translating state machines with composite states, we present in Fig. 25 the module for the normalised machine movingSTM in Fig. 17, lines `#34-#59`. That module has two program counter variables: `scpc` and `Move_scpc`, one lock variable `lock`, and two exit variables: `exit` and `Move_exit`.

The counter `scpc` encodes the current state of the machine, while `Move_scpc` encodes the current state of the composite state Move. In the normalised machine movingSTM, we have one initial junction named i0 and three states (Move, Stuck, and loop). Four constants are declared for these states (lines `#17-#20`). Extra constants declared on lines `#21-#24` are needed for the actions in transitions or states. Since Move is composite, there are additional nodes within it: one initial junction named i0 in PRISM, nine states s0 to s8, and nine probabilistic junctions named p0 to p8 in PRISM. A set of constants is declared to associate them with numbers, as illustrated in Fig. 26. We name these constants with a prefix `Move_` (see lines `#26-#31`). Like in the previous example, extra constants are also needed for the actions. Finally, we have a value, `Move_INACTIVE`, for the counter of Move, when this state is not active.

The full state of the machine is given by the combination of both counter values. For example, if the state machine movingSTM is in the state Stuck, then `scpc` and `Move_scpc` have values `Stuck` and `Move_INACTIVE`. If the machine is in the state s1, then `scpc` and `Move_scpc` have values `Move` and `Move_s1`.

In addition to the states, we also number the transitions in movingSTM, as shown in Fig. 26, with associated constants on lines `#10-#16`. We also name the transitions in Move with a prefix `Move_`. Besides `LOCK_FREE` on line `#9`, these are all the possible values that the variable `lock` can take as shown on line `#37`.

In addition, two exit variables `exit` and `Move_exit` are declared on lines `#38` and `#39`. Both can take the values 0 to 6 of the constants:

- `Exit_NONE` corresponds to EX-S1 above,
- `Exit_ACT_Parent` corresponds to EX-S2 for the request from the parent state, if any,
- `Exit_ACT_Trans` corresponds to EX-S2 for the request from a transition,
- `Exit_Sub_ACT` corresponds to EX-S3,
- `Exit_Sub_ACT_Waiting` corresponds to EX-S4,
- `Exit_Sub_EXITED` corresponds to EX-S5,
- `Exit_EXITED` corresponds to EX-S6.

In the initial state of the module `movingSTM`, the values of the local variables `scpc` and `Move_scpc` are `i0` and `Move_INACTIVE`, which indicates the machine starts from its initial junction and the composite state Move is initially inactive. The initial value of `lock` is `LOCK_FREE`. The val-

```
1  const int Exit_NONE = 0;
2  const int Exit_ACT_Parent = 1;
3  const int Exit_ACT_Trans = 2;
4  const int Exit_EXITED = 3;
5  const int Exit_Sub_ACT = 4;
6  const int Exit_Sub_ACT_Waiting=5;
7  const int Exit_Sub_EXITED = 6;
8
9  const int LOCK_FREE = 0;
10 const int Move_T0 = 1;
11 const int Move_T1 = 2;
```

```
12 ...
13 const int T0 = 11;
14 const int T1 = 12;
15 const int T2 = 13;
16 ...
17 const int i0 = 1;
18 const int Move = 2;
19 const int Stuck = 3;
20 const int loop = 4;
21 const int i0_1 = 5;
22 const int Move_1 = 6;
```

```
23 const int Move_2 = 7;
24 const int Stuck_entering=8;
25
26 const int Move_INACTIVE = 0;
27 const int Move_i0 = 1;
28 const int Move_s0 = 2;
29 const int Move_s0_entering=3;
30 const int Move_s1 = 4;
31 const int Move_s1_entering=5;
32 ...
```

```
34 module movingSTM
35   scpc : [0..8] init i0;
36   Move_scpc : [0..82] init Move_INACTIVE;
37   lock : [0..15] init LOCK_FREE;
38   exit : [0..6] init Exit_NONE;
39   Move_exit : [0..6] init Exit_NONE;
40
41   [] (scpc=i0) -> (lock'=T0)& (scpc'=i0_1);
42   [] (scpc=i0_1) -> (p'=0)& (scpc'=Move)& (Move_scpc'=Move_i0);
43
44   [] (scpc=Move)& (lock=LOCK_FREE)& (c=0)& (p!=0) -> (lock'=T1)& (Move_exit'=Exit_ACT_Trans);
45   [] (scpc=Move)& (lock=T1)& (Move_exit=Exit_EXITED) -> (scpc'=Stuck_entering);
46   [] (scpc=Stuck_entering) -> (p'=9)& (scpc'=Stuck)& (lock'=LOCK_FREE);
47
48   [] (scpc=Move)& (exit=Exit_Sub_ACT) ->
49        (Move_exit'=Exit_ACT_Parent)& (exit'=Exit_Sub_ACT_Waiting);
50   [] (scpc=Move)& (Move_exit=Exit_EXITED)& (exit=Exit_Sub_ACT_Waiting) ->
51        (exit'=Exit_Sub_EXITED)& (Move_exit=Exit_NONE);
52   [] (scpc=Move)& ((Move_exit=Exit_ACT_Parent)|(Move_exit=Exit_ACT_Trans)) ->
53        (Move_exit'=Exit_Sub_ACT);
54   [] (scpc=Move)& (Move_exit=Exit_Sub_EXITED) -> (Move_exit'=Exit_EXITED)& (Move_scpc'=Move_INACTIVE);
55
56   [] (scpc=Move)& (Move_scpc=Move_s0)& (Move_exit=Exit_Sub_ACT) -> (Move_exit'=Exit_Sub_EXITED);
57   [] (scpc=Move)& (Move_scpc=Move_s1)& (Move_exit=Exit_Sub_ACT) -> (Move_exit'=Exit_Sub_EXITED);
58   ...
59 endmodule
```

Fig. 25 The PRISM module of movingSTM

ues of both exit variables are `Exit_NONE`, so no states are being exited. In the initial state, the single transition encoded on line #41 is available. So when the value of `scpc` is `i0`, it is changed to `i0_1` and the value of `lock` is changed to `T0`. Since the value of `lock` is not `LOCK_FREE`, other transitions in the module that correspond to the transitions from a state in the machine are not available because they have a guard condition `lock=LOCK_FREE`.

When `scpc` is `i0_1`, the single transition encoded on line #42 is taken. The update changes the value of `p`, the position of the robot, to 0, which is the effect of the transition action in RoboChart, and the values of `scpc` and `Move_scpc` to `Move` and `Move_i0`.

According to the normalised machine in Fig. 17, if the current state is Move and the battery level is 0 when the robot is not in the charging station, the transition t1 from Move to Stuck in Fig. 26 is enabled. The encoding of this transition is shown on line #44. When it is taken, the module is locked in `T1` and the `exit` stage of Move is set to an exit request from a transition (`Exit_ACT_Trans`) to source state Move. The command on line #45 waits for Move to exit and is enabled only if the module is locked in `T1` and Move has been exited (`Exit_EXITED`). When an exit request is recorded in `Move_exit`, the next available command is on lines #52-#53. Since Move is composite, the update changes `Move_exit` to `Exit_Sub_ACT` to request its substates to exit first. Depending on the current state of Move, different commands are available. If the current state of Move is s0 (that is, `Move_scpc` is `Move_s0`), the command on line #56 is enabled. Since s0 has no exit action, the update simply sets the exit procedure to the next stage: the substate of Move has been exited (`Exit_Sub_EXITED`). Similarly, if the current state of Move is s1, the command on line #57

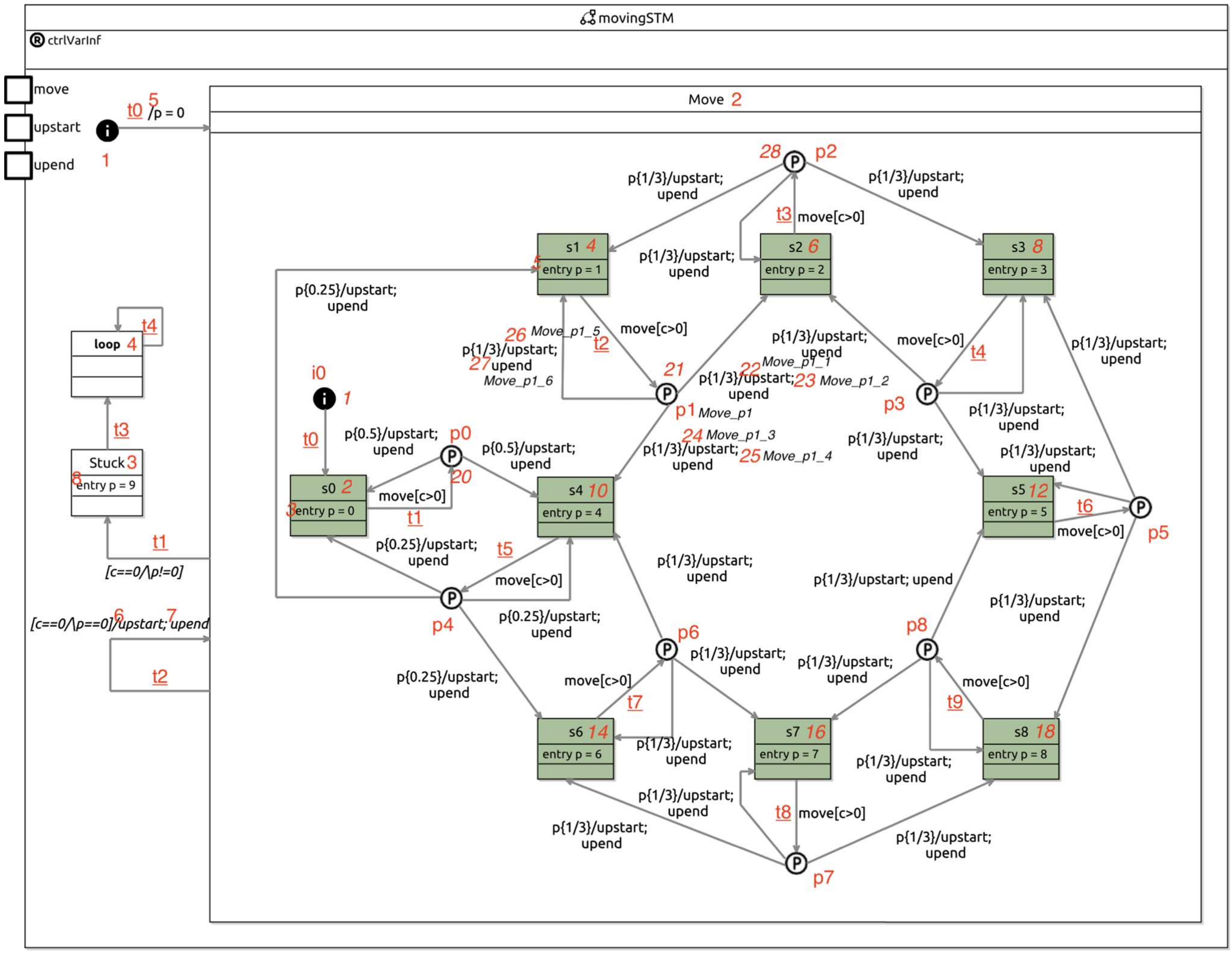


Fig. 26 Annotated normal form of movingSTM

is available. We omit similar commands for other states in Move.

After the substate is exited, the next available command is on line #54. Since Move has no exit action, the update records that the exit procedure of Move finished (Exit_EXITED), and at the same time, the counter of Move becomes Move_INACTIVE. Afterwards, the command on line #45 is available. The update changes the counter scpc to Stuck_entering, which records the start state of the entry action of Stuck. The next command on line #46 encodes the effect of the entry action (p is set to 9), the entering of the target state (the counter is at Stuck), and the completion of the transition t1 (the lock becomes LOCK_FREE).

Other transitions are encoded in a similar way, and therefore omitted in Fig. 25.

In Figs. 20, 23 and 25, we illustrate the encodings of transitions from initial junctions, states, and probabilistic junctions that have one single outgoing transition. We illustrate our approach when there are multiple transitions in Fig. 27, where we present the encodings of the incoming and outgoing transitions of the probabilistic junction p1 in Fig. 26.

The junction p1 is a node in the state Move with one incoming transition t2 from the state s1, and three outgoing transitions to the states s1, s2, and s4. Each of these outgoing transitions has an action that requires two extra states (because they are sequences of two basic actions). Along with the state for p1, there are seven states represented by the constants on lines #2-#8.

The command on lines #13-#14 encodes the transition from s1 to p1. The command on lines #17-#18 encodes the probabilistic choice and has three updates with equal probability 1/3. Each update changes the counter of Move (Move_scpc) to a state representing the start of one of the outgoing transition action. For example, the assignment in the first update changes Move_scpc to Move_p1_1, which represents the start of the action of the transition from p1 to s2 and is also annotated in Fig. 26. The commands on

```
1  ...
2  const int Move_p1 = 21;
3  const int Move_p1_1 = 22;
4  const int Move_p1_2 = 23;
5  const int Move_p1_3 = 24;
6  const int Move_p1_4 = 25;
7  const int Move_p1_5 = 26;
8  const int Move_p1_6 = 27;
9  ...
10 module movingSTM
11   ...
12   // The transition [t2] from [s1] to [p1].
13   [move] (scpc=Move)& (Move_scpc=Move_s1)& (lock=LOCK_FREE)& (c>0) ->
14        (lock'=Move_T2)& (Move_scpc'=Move_p1);
15
16   // From the probabilistic junction [p1]
17   [] (scpc=Move)& (Move_scpc=Move_p1) ->
18        (1/3):(Move_scpc'=Move_p1_1) + (1/3):(Move_scpc'=Move_p1_3) + (1/3):(Move_scpc'=Move_p1_5);
19   [upstart] (Move_scpc=Move_p1_1)& (scpc=Move) -> (Move_scpc'=Move_p1_2);
20   [upend] (Move_scpc=Move_p1_2)& (scpc=Move) -> (Move_scpc'=Move_s2_entering);
21   [upstart] (Move_scpc=Move_p1_3)& (scpc=Move) -> (Move_scpc'=Move_p1_4);
22   [upend] (Move_scpc=Move_p1_4)& (scpc=Move) -> (Move_scpc'=Move_s4_entering);
23   [upstart] (Move_scpc=Move_p1_5)& (scpc=Move) -> (Move_scpc'=Move_p1_6);
24   [upend] (Move_scpc=Move_p1_6)& (scpc=Move) -> (Move_scpc'=Move_s1_entering);
25
26   [] (Move_scpc=Move_s1_entering)& (scpc=Move) -> (p'=1)& (Move_scpc'=Move_s1)& (lock'=LOCK_FREE);
27   ...
28 endmodule
```

Fig. 27 The PRISM encoding of transitions from probabilistic junctions in movingSTM

lines #19-#20, #21-#22, and #23-#24 are the encodings of the three transition actions in PRISM. In the end of each transition action, the counter is set to an extra state representing the start of the entry action of the target state, such as Move_s1_entering for s1. The entry action of s1 is encoded by the command on line #26.

Next, we present the normalisation rules.

5.3 Normalisation rules

In terms of model transformations, normalisation is characterised as a rule-based, homogeneous, and declarative transformation. We present here a set of functions from a RoboChart model that satisfies our translation requirements to a RoboChart model in normal form. These functions are defined via rules.

To normalise a RoboChart model, we apply Rule 1 to each state machine. This is the rule applied exhaustively to all state machines in the normalisation process discussed previously. In the definition of all rules, we use the constructs from Z [70,72] as a meta-notation. Their syntax is summarised in Table 3.

Generally, a rule characterises a function on elements of the RoboChart metamodel. A rule definition is composed of a declaration, a body, and a **where** clause. The declaration gives the function name, its parameters (names and types), and its return type. In Rule 1, the function is $[\![-]\!]_{\mathcal{STM}}$. It has one parameter stm of type StateMachineDef, that is, a state machine, and also returns a state machine: an element of the same type.

The body of a rule defines an object of the return type. For example, the body of Rule 1 is an object of the class StateMachineDef of the metamodel.

The variables used in the body may be parameters or may be defined in the **where** clause. The order of these definitions is not relevant.

The body of Rule 1 defines a state machine whose name, variable list, operations, events, and interfaces are the same as those of the parameter stm. The nodes and transitions, however, are different and are specified using the definitions from the **where** clause. The first four definitions are related to the four steps of the state machine normalisation process presented above.

- ▶ (loopstates, looptrans) is a pair characterised by applying addLoopStateTrans to stm (see Rule 9 in "Appendix A"). This function is concerned with the possible need to introduce loop states. The result of addLoopStateTrans(stm) is a pair, whose first element loopstates is a set either empty or containing a new loop state for stm, and whose second element looptrans is a set of the new transitions to and from loop.
- ▶ (remjuncs, deltrans, newtrans) is a triple of sets: a set of junctions, and two sets of transitions defined

by an application of the function combTransJunctions. This function is concerned with the combination of incoming and outgoing transitions of normal junctions, if needed. The first argument of the application is a set of normal junctions, characterised by a function inTransCombinableJuncs (see Rule 10 in "Appendix A"). These are the normal junctions that have incoming and outgoing transitions that need to be combined. The second argument of the application is the set of transitions of stm that are to or from junctions in the first argument; this is characterised by a generalised union of a relational image. The function application transitionsOf (stm.transitions) identifies the set of incoming and outgoing transitions of a given node n that are in the set of transitions stm.transitions. So the relational image of inTransCombinableJuncs(stm) under transitionsOf(stm.transitions) is the set of sets of all incoming and outgoing transitions of the normal junctions. A generalised union $\bigcup$ combines all these sets. The first and seconds elements remjuncs and deltrans of the result of the application of combTransJunctions contain the junctions in the first argument and the transitions in the second argument removed by normalisation, and the third element newtrans contains the new transitions arising from the normalisation.

▶ transpjunc is a set of pairs defined by applying splitTran (Rule 13 in "Appendix A") to each transition in stm (stm.transitions), to the new transitions identified by combTransJunctions (that is, in newtrans), and to the transitions from the new loop states (identified by looptrans), but not to the transitions removed by the normalisation (that is, in deltrans).

▶ compstates is a set of normalised composite states resulting from the application of a normalisation function $[\![-]\!]_{\mathcal{S}}$ for states to each node n of the machine that is a state ($\mathsf{n} \in \mathsf{States}$) and has nodes itself ($\#(\mathsf{n.nodes}) > 0$). These are the composite states. Normalisation of composite states by $[\![-]\!]_{\mathcal{S}}$ is similar to that of state machines except that composite states do not declare variables, operations, and events. So the formalisation is simpler and omitted here; it can be found in the RoboChart reference manual [1].

▶ intactnodes is the set of nodes n of stm that are neither the normal junctions removed by the normalisation in combTransJunctions (that is, remjuncs) nor composite states. They are unaffected by normalisation.

The nodes of the normalised machine, as defined in the body of Rule 1, include the loop states in loopstates, the new probabilistic junctions in the second element of each pair in transpjunc (obtained using generalised union $\bigcup$), the normalised composite states, and all other nodes as identified in intactnodes. The transitions of the normalised machine are just those in the first element of each pair in transpjunc.

Next, we present the second stage of translation from a RoboChart model in normal form to PRISM.

5.4 Transformation to PRISM

In this section, we present the translation rules used in our approach in Sects. 5.4.1 to 5.4.4.

Rule 1. Normalisation of state machines $[\![\mathsf{stm} : \mathsf{StateMachineDef}]\!]_{\mathcal{STM}} : \mathsf{StateMachineDef} =$

$$\left\langle \begin{array}{l} \mathsf{name} \rightsquigarrow \mathsf{stm.name},\ \mathsf{variableList} \rightsquigarrow \mathsf{stm.variableList},\ \mathsf{operations} \rightsquigarrow \mathsf{stm.operations},\ \mathsf{events} \rightsquigarrow \mathsf{stm.events}, \\ \mathsf{pInterfaces} \rightsquigarrow \mathsf{stm.pInterfaces},\ \mathsf{rInterfaces} \rightsquigarrow \mathsf{stm.rInterfaces},\ \mathsf{interfaces} \rightsquigarrow \mathsf{stm.interfaces}, \\ \mathsf{nodes} \rightsquigarrow (\mathsf{loopstates} \cup (\bigcup\{\mathsf{tp} : \mathsf{transpjunc} \bullet \mathsf{tp.2}\}) \cup \mathsf{compstates} \cup \mathsf{intactnodes}), \\ \mathsf{transitions} \rightsquigarrow \bigcup\{\mathsf{tp} : \mathsf{transpjunc} \bullet \mathsf{tp.1}\} \end{array} \right\rangle_{\mathsf{StateMachineDef}}$$

where

$$\begin{array}{l} (\mathsf{loopstates}, \mathsf{looptrans}) = \mathsf{addLoopStateTrans}(\mathsf{stm}) \\ (\mathsf{remjuncs}, \mathsf{deltrans}, \mathsf{newtrans}) = \\ \qquad \mathsf{combTransJunctions} \left(\begin{array}{l} \mathsf{inTransCombinableJuncs}(\mathsf{stm}), \\ \bigcup (\mathsf{transitionsOf}(\mathsf{stm.transitions}) (\!| \mathsf{inTransCombinableJuncs}(\mathsf{stm}) |\!)) \end{array} \right) \\ \mathsf{transpjunc} = \{\mathsf{t} : \mathsf{stm.transitions} \cup \mathsf{newtrans} \cup \mathsf{looptrans} \setminus \mathsf{deltrans} \bullet \mathsf{splitTran}(\mathsf{t})\} \\ \mathsf{compstates} = \{\mathsf{n} : \mathsf{stm.nodes} \mid \mathsf{n} \in \mathsf{State} \land \#(\mathsf{n.nodes}) > 0 \bullet [\![\mathsf{n}]\!]_{\mathcal{S}}\} \\ \mathsf{intactnodes} = \{\mathsf{n} : \mathsf{stm.nodes} \mid \mathsf{n} \notin \mathsf{remjuncs} \land \neg\ (\mathsf{n} \in \mathsf{State} \land \#(\mathsf{n.nodes}) > 0)\} \end{array}$$

5.4.1 Module, robotic platform, and controllers

Rule 2. Module $[\![\mathsf{m} : \mathsf{Module}]\!]_{\mathcal{M}} : \mathsf{Model}_{\mathsf{pr}} =$

$$\left\langle \begin{array}{l} \mathsf{constants} \rightsquigarrow (\bigcup\{\mathsf{r} : \mathsf{ctrlrets} \bullet \mathsf{r.1}\}) \cup \mathsf{rpconsts} \cup \mathsf{exitconsts}, \\ \mathsf{globals} \rightsquigarrow (\bigcup\{\mathsf{r} : \mathsf{ctrlrets} \bullet \mathsf{r.2}\}) \cup \mathsf{rpvars}, \\ \mathsf{modules} \rightsquigarrow (\bigcup\{\mathsf{r} : \mathsf{ctrlrets} \bullet \mathsf{r.3}\}) \cup \{\mathsf{rpmodule}\} \end{array} \right\rangle_{\mathsf{Model_{pr}}}$$

where

$$\begin{array}{l} \mathsf{ctrlrets} = [\![-]\!]_{\mathcal{C}} \,(\!| \{\mathsf{n} : \mathsf{m.nodes} \mid \mathsf{n} \in \mathsf{Controller}\} |\!) \\ \mathsf{rpoutevents} = \{\mathsf{c} : \mathsf{m.connections} \mid \mathsf{c.from} \in \mathsf{RoboticPlatform} \wedge \mathsf{c.efrom.type} \neq \mathsf{null} \bullet \mathsf{c.efrom}\} \\ (\mathsf{rpconsts}, \mathsf{rpvars}, \mathsf{rpmodule}) = [\![(\mu\, \mathsf{n} : \mathsf{m.nodes} \mid \mathsf{n} \in \mathsf{RoboticPlatform})\,, \mathsf{rpoutevents}, (\bigcup\{\mathsf{r} : \mathsf{ctrlrets} \bullet \mathsf{r.4}\})]\!]_{\mathcal{R}} \\ \mathsf{exitconsts} = \mathsf{exitSeqCtrlConsts}() \end{array}$$

The main translation function $[\![-]\!]_{\mathcal{M}}$ (defined in Rule 2) is for a RoboChart module. It has a parameter m of type Module, and characterises a PRISM model of type Model constructed from constants, global variables globals, and modules from the translation of the controllers and the robotic platform of m.

In this section, we use the following conventions for function definitions in rules: the types of the arguments are assumed to be classes of the RoboChart metamodel and the types of the results are assumed to be classes of the PRISM metamodel if no subscripts are specified. For example, the type of m in the definition of the function $[\![-]\!]_{\mathcal{M}}$, Module, is the class in the RoboChart. Particularly, if a class in the PRISM metamodel is used as a type for arguments, or a class in the RoboChart metamodel is used as a type for results, we use an explicit subscript to indicate it.

The result of the translation of the controllers is recorded in a set of quadruples ctrlrets, while that of the robotic platform is in a triple (rpconsts, rpvars, rpmodule). Each quadruple in ctrlrets records information about the translation of a controller. The first element of each quadruple r in ctrlrets, r.1, is a set of constants resulting from the translation of a controller. So the constants for the all controllers of m are combined using generalised union $\left(\bigcup\{\mathsf{r} : \mathsf{ctrlrets} \bullet \mathsf{r.1}\}\right)$ and contributed to constants. The second and third elements of r, r.2 and r.3, are sets of global variables and modules. They are combined using generalised union and contributed to globals and modules. Accordingly, rpconsts, rpvars, and rpmodule are a set of constants, a set of variables, and a module, contributed to constants, globals, and modules.

Additionally, a set of constants exitconsts is used to specify the control flow of execution of a state machine with composite states; in particular, the constants are useful to define the flow when exiting a composite state. This is discussed later on in Sect. 5.4.2.

The variables ctrlrets, rpconsts, rpvars, rpmodule, and exitconsts are defined in the **where** clause of Rule 2.

► ctrlrets is a set of quadruples: sets of constants, variables, and modules, and a relation defined by an application of $[\![-]\!]_{\mathcal{C}}$ (Rule 4) to a controller in the set m.nodes of nodes of m. The relation establishes a mapping from operations provided by the platform and required by the controller, to PRISM actions corresponding to these operations.

► rpoutevents is used in the translation of the robotic platform. It is the set of events of the robotic platform used to communicate data to a controller. These are the events c.efrom that are the source of a connection c in the set m.connections of connections of m whose source c.from is a platform and whose type c.efrom.type is not null. So, data (of type c.efrom.type) is communicated.

► (rpconsts, rpvars, rpmodule) is a triple: a set of constants, a set of variables, and a module defined by an application of $[\![-]\!]_{\mathcal{R}}$ (Rule 3) for robotic platforms. This function is applied to the platform of m, identified by a unique (μ) node n that is in the set m.nodes of nodes of m and is a robotic platform. Extra arguments are the set of events rpoutevents and the relation from the translation of controllers, formed by generalised union of the relation r.4 for each controller. As said, the events used to send data to a controller are represented by a shared variable in the platform module. (Its value is nondeterministically chosen and copied in modules for state machines upon synchronisation.) So, the definition of the module for a platform requires the identification of these events, which are in the set rpoutevents. The relation identifies how the translation of the controllers (or, more specifically, of their state machines) have mapped their required operations to PRISM actions. The module for the platform synchronises on these actions.

► exitconsts is a set of constants resulting from the function exitSeqCtrlConsts, omitted here.

In Fig. 19, the result of the translation of the unique controller of our example gives rise to the declaration of `NoTask` in the constants section and all modules except deliverRP. The declarations of two constants and two global variables in the constants and global sections and the module `deliverRP` are the result of the translation of the robotic platform. The constant `NoTask` is the result of the translation of the enumerated type FD in Fig. 4. More constants result from the translation of the controller, but are omitted in Fig. 19. The details are presented in Sects. 5.4.2 and 5.4.3.

Figure 19 gives the module for deliverRP, and corresponding constants and global variables provided by it. As shown in Fig. 2, deliverRP provides two constants and two variables. Their counterparts in PRISM are two constants and two global variables, shown on lines `#4`-`#7`, of type int. The platform has two output events request and dest as identified from connections in deliverMOD in Fig. 2. Both events are of type nat, natural numbers. For each output event, we add an extra local variable of a corresponding type, such as EVT__request (on line `#10` in Fig. 19) for request, and a set of commands with a corresponding action for the event name such as the commands indicated on lines `#13`-`#15`, to the module. Together, the commands characterise a nondeterministic choice of the value of the variable. Other events used for output are handled in a similar way.

Rule 3. Robotic platforms

$$[\![\mathsf{rp} : \mathsf{RoboticPlatformDef}, \mathsf{outevents} : \mathbb{P}\ \mathsf{Event}, \mathsf{opmaps} : \mathsf{OperationSig} \leftrightarrow \mathsf{Action}_{\mathsf{pr}}]\!]_{\mathcal{R}}$$
$$: \mathbb{P}\ \mathsf{Constant} \times \mathbb{P}\ \mathsf{VarDecl} \times \mathsf{Module} =$$

$$\left(\bigcup \{\mathsf{r} : \mathsf{constvars} \bullet \mathsf{r.1}\}, \bigcup \{\mathsf{r} : \mathsf{constvars} \bullet \mathsf{r.2}\}, \mathsf{module}\right)$$
$$\mathbf{where}$$
$$\mathsf{constvars} = [\![\_]\!]_{\mathcal{VL}} (\!|\ \mathsf{getVariableLists}(\mathsf{rp})\ |\!)$$
$$\mathsf{localvars} = \{\mathsf{e} : \mathsf{outevents} \bullet \mathtt{EVT\_\_}\mathsf{id}(\mathsf{e}) : [\![\mathsf{e.type}]\!]_{t}\ ;\ \}$$
$$\mathsf{eventcmds} = \left\{\mathsf{e} : \mathsf{getEvents}(\mathsf{rp}) \bullet \left(\begin{array}{l}\mathbf{if}\ \ \mathsf{e} \in \mathsf{outevents}\ \mathbf{then}\ \{\mathsf{v} : [\![\mathsf{e.type}]\!]_{t} \bullet [\,\mathsf{id}(\mathsf{e})\,]\,\mathtt{true} \rightarrow \mathtt{EVT\_\_}\mathsf{id}(\mathsf{e})' = \underline{\mathsf{v}}\ ;\ \} \\ \mathbf{else}\ \ \{[\,\mathsf{id}(\mathsf{e})\,]\,\mathtt{true} \rightarrow \mathtt{true}\ ;\ \}\end{array}\right)\right\}$$
$$\mathsf{opcmds} = \{\mathsf{act} : \mathrm{ran}\ \mathsf{opmaps} \bullet [\,\mathsf{act.name}\,]\,\mathtt{true} \rightarrow\ \mathtt{true}\ ;\ \}$$
$$\mathsf{module} = \mathtt{module}\ \mathsf{id}(\mathsf{rp}), \mathsf{vars} \rightsquigarrow \mathsf{localvars}, \mathsf{commands} \rightsquigarrow \left(\bigcup \mathsf{eventcmds}\right) \cup \mathsf{opcmds}\ \mathtt{endmodule}$$

As mentioned, $[\![-]\!]_{\mathcal{R}}$ defines a module like deliverRP; it has three parameters: a robotic platform rp, a set outevents of events of rp used for output, and a relation opmaps that maps operations provided by rp to PRISM actions. In Rule 3, we note the type of rp is RoboticPlatformDef, not RoboticPlatform used in Rule 2. The class RoboticPlatform is inherited by both RoboticPlatformDef and RoboticPlatformRef. So the robotic platform identified in Rule 2 is either a RoboticPlatformDef or a RoboticPlatformRef object. For a RoboticPlatformRef rp, another rule specifies its result just as the function application $[\![\mathsf{rp.ref}]\!]_{\mathcal{R}}$ (the unique identifiers defined by id(n) - see Table 3 - take references into account). Rules for ControllerRef and StateMachineRef are similar and omitted here.

The result of $[\![\mathsf{rp}, \mathsf{outevents}, \mathsf{opmaps}]\!]_{\mathcal{R}}$ is a triple, whose first and second elements are a set of constants ($\bigcup \{\mathsf{r} : \mathsf{constvars} \bullet \mathsf{r.1}\}$ where r is a pair of sets of constants and variables) and a set of variables ($\bigcup \{\mathsf{r} : \mathsf{constvars} \bullet \mathsf{r.2}\}$) defined by the constants and variables constvars (a set of pairs) provided by rp, and whose third element is the PRISM module corresponding to rp. The variables constvars and module are defined in the **where** clause of Rule 3 and described below.

- ▶ constvars is a set of pairs of sets. The first and second elements of each pair are a set of constants and a set of variables, translated from each variable list vl in rp by the function $[\![\mathsf{vl}]\!]_{\mathcal{VL}}$ (omitted here). The lists of variables provided by rp are identified using the function getVariableLists whose definition is omitted here. A variable list consists of a set of constant and variable declarations. The translation of variable declarations maps the RoboChart types to the PRISM types, and translates the initial value given, if any, accordingly.
- ▶ localvars is a set of PRISM declarations of variables like EVT__request corresponding to the events e from outevents. In the rules, we use black typewriter font, like in `true`, for PRISM terms. They are abbreviations for instances of the classes of the PRISM metamodel presented in Sect. 4.2. For clarity, however, we use the PRISM concrete syntax to represent them. Each declaration in localvars is for a variable, whose name is based on the name of e in PRISM (id(e)) with a prefix (`EVT__`), and whose type $[\![\mathsf{e.type}]\!]_{\mathsf{t}}$ is obtained by translating the type e.type of e using $[\![-]\!]_{\mathsf{t}}$.
- ▶ eventcmds is a set of sets of commands for each event e of rp, that are identified using the function getEvents whose definition is omitted here. If e is an output event (that is, e ∈ outevents), a set of commands is included in eventcmds to define a nondeterministic choice of the values for the corresponding variable `EVT__`id(e). Each command corresponds to one possible value v in the event type $[\![\mathsf{e.type}]\!]_{\mathsf{t}}$. The action of the command (inside square brackets) is the name id(e) of the event e, the guard is `true`, and the only update (between → and `;`) assigns v to `EVT__`id(e). The commands on lines `#13`-`#15` in Fig. 19 are obtained in this way. If e is not an event

for output, a set containing one command is included in eventcmds. It has id(e) for its action and `true` for both its guard and its updates, which means the command is always ready for synchronisation on the action and no variable is changed upon synchronisation. The command on line `#18` is obtained in this way.

▶ opcmds is a set of PRISM commands. Each command corresponds to one PRISM action act in the range ran opmaps of the relation opmaps. As previously mentioned, opmaps records a mapping from operations provided by the robotic platform (and possibly called in state machines) to PRISM actions corresponding to these operations. We use this relation to translate calls to the operations in state machines into communications with the environment (modelled by the module for the robotic platform) that provides the operations. Our PRISM models, as explained in Sect. 4, rely on the name of actions for synchronisation. This imposes extra challenges on modelling of operations in PRISM because the calls to a same operation from different state machines must not be synchronised. We consider, for instance, an example in which an operation moveSteps with a parameter of type int is provided by the robotic platform, and required and called using an action moveSteps(1) in a state machine stm1 and moveStep(2) in another state machine stm2. The operation calls are translated to communications in PRISM through synchronisation on actions with assistance of variables. If two calls were mapped into an action of the same name such as moveSteps_act, then both state machines and the robotic platform would synchronise on this action, which is not the right semantics of operation calls. For this reason, we allocate a unique action name for an operation in each state machine, such as stm1_moveSteps_act for stm1 and stm2_moveSteps_act for stm2. So in the platform, an operation is mapped to several actions: one for each machine that requires the operation. For this example, opmaps is given below.

Rule 4. Controllers $[\![\mathsf{ctrl} : \mathsf{ControllerDef}]\!]_{\mathcal{C}} : \mathbb{P}\ \mathsf{Constant} \times \mathbb{P}\ \mathsf{VarDecl} \times \mathbb{P}\ \mathsf{Module} \times (\mathsf{OperationSig} \leftrightarrow \mathsf{Action}) =$

$$\left(\bigcup \{\mathsf{r} : \mathsf{constvars} \bullet \mathsf{r.1}\} \cup \left(\bigcup \{\mathsf{r} : \mathsf{stmrets} \bullet \mathsf{r.1}\}\right), \bigcup \{\mathsf{r} : \mathsf{constvars} \bullet \mathsf{r.2}\}, \{\mathsf{r} : \mathsf{stmrets} \bullet \mathsf{r.2}\}, \mathsf{rops}\right)$$

$$\begin{array}{l} \mathbf{where} \\ \quad \mathsf{constvars} = [\![\_]\!]_{\mathcal{VL}} (\!|\ \mathsf{getVariableLists(ctrl)}\ |\!) \\ \quad \mathsf{stmrets} = \left\{ \begin{array}{l} \mathsf{stm} : \mathsf{ctrl.machines} \bullet \\ \quad \mathbf{let}\ \mathsf{stmoutevents} == \{\mathsf{c} : \mathsf{ctrl.connections} \mid \mathsf{c.from} = \mathsf{stm} \wedge \mathsf{c.efrom.type} \neq \mathsf{null} \bullet \mathsf{c.efrom}\} \\ \qquad\quad \bullet [\![\mathsf{stm}, \mathsf{stmoutevents}]\!]_{\mathcal{STM}} \end{array} \right\} \\ \quad \mathsf{rops} = \bigcup \{\mathsf{stm} : \mathsf{ctrl.machines} \bullet \mathsf{op2ActionMaps(stm)}\} \end{array}$$

$$\left\{ \begin{array}{l} (moveSteps, stm1\_moveSteps\_act)\,, \\ (moveSteps, stm2\_moveSteps\_act) \end{array} \right\} \tag{1}$$

Since the mapping is established in the state machines, the relation is a result of the translation of controllers (which involves translating their machines) and passed to $[\![\_]\!]_{\mathcal{R}}$. For each action act in the range of opmaps (for our example, $\{stm1\_moveSteps\_act, stm2\_moveSteps\_act\}$), one command, whose action is act, is included in opcmds. This command in the platform module synchronises with the commands in a machine module whose action is also act, to allow the environment to record the calls to the operation. The arguments of these calls are recorded in local variables of the machine modules. The details of the declaration and use of these variables are presented in Rule 5. Both the guard and the updates of the command are `true`, so the platform module cannot refuse synchronisation on act and no variable is changed upon synchronisation. For our example, opcmds contains the commands in Fig. 28. They allow the platform module to record the call to moveSteps from both stm1 and stm2, though in different actions.

▶ module is a PRISM module with the name id(rp) of rp, localvars, and commands obtained by the generalised union of the commands in eventcmds and opcmds.

The modules in Fig. 19 other than deliverRP are defined by the function $[\![\_]\!]_{\mathcal{C}}$ for a controller ctrl Rule 4. The result is a quadruple: a set of constant declarations, a set of variable declarations, a set of modules, and a relation from the operations provided by the platform and required by the machines in the controller to corresponding PRISM actions. For the moveSteps example, if both stm1 and stm2 are in ctrl, then the resulting relation is the one shown in (1). Analogous to Rule 3, we define constvars in the **where** clause to record the constants and variables provided by ctrl.

```
[ stm1_moveSteps_act ] true -> true;
[ stm2_moveSteps_act ] true -> true;
```

Fig. 28 The commands in opcmds

The results of the translation of all state machines of ctrl are recorded in a set of pairs stmrets, also defined in the **where** clause. Each pair corresponds to one state machine and includes a set of constants and a module. The constants in these pairs are combined using generalised union and are also contributed to the result of $\llbracket - \rrbracket_{\mathcal{C}}$. Additionally, the relation is recorded in rops.

op2ActionMaps takes one argument: a state machine stm in the set of machines ctrl.machines of ctrl.

Next, we describe how to translate state machines.

5.4.2 State machines and composite states

Here, we present the rule that can be used to translate a RoboChart state machine to PRISM.

Rule 5. State machines

```
  ⟦stm : StateMachineDef, outevents : ℙ Event⟧_STM : ℙ Constant × Module =

      ⋃{r : constvars • r.1};
      const int id(stm)_LOCK_FREE = 0;
      stmconsts;
      module id(stm)
        id(stm)_lock : [0..(trnumber − 1)] = id(stm)_LOCK_FREE;
        ⋃{r : constvars • r.2};
        outeventvars;
        stmvars;
        ropsvars;
        stmcmds;
      endmodule
  where
      constvars = ⟦_⟧_VL ⦇ getVariableLists(stm) ⦈
      outeventvars = {e : outevents • EVT__id(e) : ⟦e.type⟧_t ; }
      (trnumber, stmconsts, stmvars, stmcmds) = ⟦stm, stm, ∅, 1⟧_S
      ropsvars = ⋃{ op : getRequiredOperations(stm) • {p : op.parameters • uname (stm, op.name, p.name) : ⟦p.type⟧_t ; } }
```

Each pair in stmrets is the result of the translation of one state machine stm in the set of machines ctrl.machines of ctrl by the function $\llbracket - \rrbracket_{\mathcal{STM}}$ (Rule 5), as defined in the **where** clause. The function application takes two arguments: the state machine stm and a set stmoutevents of events of stm used for output. The set stmoutevents includes events of the state machine used to communicate data to the controller or to another state machine in the controller. These are the events c.efrom that are the source of a connection c in the set ctrl.connections of connections of ctrl whose source c.from is stm and whose type c.efrom.type is not null. So data (of type c.efrom.type) is communicated. These events are used in the definition of $\llbracket - \rrbracket_{\mathcal{STM}}$ to identify variables that need to be included in the module for stm to assist communications. We consider, for instance, a state machine stm1 with one event out used for output. So stmoutevents, in this case, has one element out. The module for stm1, translated by $\llbracket$ stm1, {out}$\rrbracket_{\mathcal{STM}}$, therefore has a local variable `EVT__out` of the same type as out for the purpose of communications.

The relation rops is also defined in the **where** clause. It is the result of combining the relations established for state machines by the function op2ActionMaps (Rule 14 in "Appendix A"), using generalised union. The function

Each machine module in Fig. 19, such as taskSTM and movingSTM in Figs. 20 and 25, is defined by the function $\llbracket - \rrbracket_{\mathcal{STM}}$ in Rule 5. As mentioned previously, this function is concerned with the translation of a state machine stm, having a set outevents of events of stm used for output as an extra parameter. Its result is a pair: a set of constants and a module.

Analogous to Rule 3, we define constvars as a set of pairs to record the constants and variables required by or defined in stm. The first element r.1 of each pair r in constvars is a set of constant declarations and the second element r.2 is a set of variable declarations. Generalised union $\bigcup$ combines all these sets. A constant, named id(stm)`_LOCK_FREE`, such as `LOCK_FREE` on line `#5` in Fig. 20, is associated with 0 and results from the translation of stm. Rule 5 uses id(stm) as a prefix for the constant or variable names to ensure unicity. Other constants, such as those on lines `#5`-`#10`, for the values that `lock` can take, and on lines `#12`-`#23`, for the states in `taskSTM`, are recorded in stmconsts.

The machine module is named id(stm) and contains variable declarations and commands. The variables include `lock`, whose type is an interval with upper limit trnumber − 1, such as that on line `#27` in Fig. 20; the local variables of stm, like `fd` and `g` on lines `#28`-`#29`, as recorded in the second element r.2 of pairs r in constvars, and combined using generalised union; the variables to encode communication through the events in outevents, in outeventvars; the scpc and exit variables, like those on lines #35-#39 in Fig. 25, as recorded in stmvars; and the variables to encode the arguments of calls to the operations required by stm, in ropsvars. The set of commands is recorded in stmcmds.

The **where** clause in the rule gives more details.

- ▶ outeventvars is a set of PRISM declarations of variables (similar to EVT__request for the robotic platform) corresponding to the events e from outevents. Each declaration in outeventvars is for a variable whose name is based on the name of e in PRISM (id(e)) with a prefix (EVT__), and whose type $[\![\mathsf{e.type}]\!]_{\mathsf{t}}$ is obtained by translating the type e.type of e using the function $[\![-]\!]_{\mathsf{t}}$.
- ▶ (trnumber, stmconsts, stmvars, stmcmds) is a quadruple of an integer, and sets of constants, variables, and commands, resulting from the application of $[\![-]\!]_{\mathcal{S}}$ (Rule 6) to stm to translate its nodes and transitions. This function is concerned with the translation of a NodeContainer (a state machine or composite state) given as its first argument, with the machine that owns the NodeContainer as its second argument. An extra third argument is a set of boolean expressions that records the containers for the node and defines the guard for the commands that encode the node. The guard is the conjunction of these expressions, which are equalities involving the counter variables that, together, indicate that the control flow has reached the node. For example, to translate Move in movingSTM using this function, the right argument is {scpc=Move} to indicate the control flow has reached Move. If Move has another composite substate S, then the right argument to translate S is {scpc=Move, Move_scpc=Move_S} to indicate the control flow has reached both Move and S. The corresponding guard is scpc=Move & Move_scpc=Move_S, the conjunction of both expressions. A final argument for $[\![-]\!]_{\mathcal{S}}$ indicates the first integer not yet used to declare a constant for a lock value. In Rule 5, $[\![-]\!]_{\mathcal{S}}$ is applied to the whole machine stm, which, by convention, is owned by itself. The third argument is just $\emptyset$, because the control flow is always in the machine. Finally, the final argument indicates that declarations of lock variables can use integers from 1. (We recall that 0 is used for the LOCK_FREE constant.) As a result of this function application, trnumber is the next available number that can be used to number a transition. So the maximum transition number is trnumber − 1, which is used for the upper limit of the interval type of the lock variable. For the taskSTM module in Fig. 20, the next available transition number is 6 because the numbers 1 to 5 are used in the declarations on lines #5-#10. The second and third elements identify sets of constant and variable declarations, like those on lines #6-#23 and the scpc declaration on line #26. These are the constants and variables used in the encoding of the control flow of the node. The fourth element is the set of all commands in the module.
- ▶ ropsvars is a set of variable declarations for each parameter p of an operation op required by stm. The variable has a fresh name uname(stm, op.name, p.name) and is of type $[\![\mathsf{p.type}]\!]_{\mathsf{t}}$ obtained by translating the type p.type of p using the function $[\![-]\!]_{\mathsf{t}}$. For the operation moveSteps with parameter steps of type int, the value of ropsvars is $\left\{\texttt{stm\_moveSteps\_steps:}\ [\![\mathsf{int}]\!]_{\mathsf{t}}\,\texttt{;}\right\}$.

Rule 6. Composite states

$$
\begin{array}{l}
[\![\mathsf{cs : NodeContainer, stm : StateMachineDef, pcconstrs} : \mathbb{P}\ \mathsf{BoolExpr_{pr}, trnumber : int}]\!]_{\mathcal{S}} \\
\quad : \mathsf{int} \times \mathbb{P}\ \mathsf{Constant} \times \mathbb{P}\ \mathsf{VarDecl} \times \mathbb{P}\ \mathsf{Command} =
\end{array}
$$

$$
\begin{array}{l}
\left(\begin{array}{l}
\mathsf{sstrnumber}, \{\mathsf{const0}\} \cup \mathsf{exitcsconsts} \cup \mathsf{exitsubconsts} \cup \mathsf{entersubconsts} \cup \mathsf{nodesconsts} \cup \mathsf{tnsconsts} \cup \mathsf{ssconsts}, \\
\{\mathsf{scpc, exit}\} \cup \mathsf{ssvars}, \mathsf{exitcscmds} \cup \mathsf{exitsubcmds} \cup \mathsf{entersubcmds} \cup \mathsf{tnscmds} \cup \mathsf{sscmds}
\end{array}\right) \\
\mathbf{where} \\
\quad \mathsf{const0} = \texttt{const int}\ \mathsf{id(cs)\_INACTIVE} = 0; \\
\quad \mathsf{i0} = \mu\, \mathsf{n : cs.nodes} \bullet \mathsf{n} \in \mathsf{Initial} \\
\quad \mathsf{scpc} = \mathsf{id(cs)\_scpc} : [0\,..\,(\mathsf{tnsstnumber} - 1)] = \mathbf{if}\ \mathsf{cs} \in \mathsf{StateMachineDef}\ \mathbf{then}\ \mathsf{id(i0)}\ \mathbf{else}\ \mathsf{const0.name}; \\
\quad \mathsf{exit} = \mathsf{id(cs)\_exit} : [0\,..\,6] = \mathsf{Exit\_NONE}; \\
\quad (\mathsf{exitcsstnumber, exitcsconsts, exitcscmds}) = \\
\qquad \mathbf{if}\ \mathsf{cs} \in \mathsf{StateMachineDef}\ \mathbf{then}\ (1, \emptyset, \emptyset)\ \mathbf{else}\ \mathsf{exitCompState}\,(\mathsf{cs, exit, id(cs)}\texttt{\_scpc}, \mathsf{pcconstrs}, 1) \\
\quad (\mathsf{exitsubstnumber, exitsubconsts, exitsubcmds}) = \\
\qquad \mathsf{exitSubstates}\,(\{\mathsf{s : cs.nodes} \mid \mathsf{s} \in \mathsf{State}\}, \mathsf{exit, id(cs)}\texttt{\_scpc}, \mathsf{pcconstrs, exitcsstnumber}) \\
\quad (\mathsf{entersubstnumber, entersubconsts, entersubcmds}) = \\
\qquad \mathsf{enterSubstates}\left(\begin{array}{l}
\{\mathsf{s : cs.nodes} \mid \mathsf{s} \in \mathsf{State} \wedge \mathsf{getEntryAction}\,(\mathsf{s}) \neq \mathsf{null}\}, \\
\mathsf{id(cs)}\texttt{\_scpc}, \mathsf{pcconstrs, exitsubstnumber}
\end{array}\right) \\
\quad (\mathsf{nodesstnumber, nodesconsts}) = \mathsf{constantsOfNamedElems}\,(\mathsf{cs.nodes, entersubstnumber}) \\
\quad (\mathsf{tnsstnumber, tnstrnumber, tnsconsts, tnscmds}) = \\
\qquad [\mathsf{cs.nodes, cs, stm, exit, id(cs)}\ \texttt{\_scpc}, \mathsf{pcconstrs, nodesstnumber, trnumber}]_{\mathcal{TNS}} \\
\quad (\mathsf{sstrnumber, ssconsts, ssvars, sscmds}) = \\
\qquad [\{\mathsf{s : cs.nodes} \mid \mathsf{s} \in \mathsf{State} \wedge \mathsf{isComposite(s)}\}, \mathsf{stm, id(cs)}\ \texttt{\_scpc}, \mathsf{pcconstrs, tnstrnumber}]_{\mathcal{SS}}
\end{array}
$$

In Rule 6, the parameters for $[\![-]\!]_{\mathcal{S}}$ are a NodeContainer cs, the state machine stm containing cs, a set of PRISM expressions pcconstrs recording constraints on the program counters, and the next number trnumber that can be used to define a constant for a transition.

As illustrated in Fig. 25, the translation of a NodeContainer cs such as Move introduces a) one program counter variable and one exit variable; b) the encoding for exiting cs, such as the commands on lines `#52-#54`; c) the encoding for exiting the substates of cs, such as the commands on lines `#48-#51`, `#56`, and `#57`; d) the encoding for entering the substates of cs that have entry actions, such as the command on line `#46`; e) the encoding for the transitions of cs, such as the commands on lines `#42`; and f) the encoding for the composite substates of cs. There may also be constant declarations, such as those on lines `#9-#31`.

In the definition of $[\![-]\!]_{\mathcal{S}}$, the declarations of the program counter and the exit variable are recorded in scpc and exit. The encoding for exiting cs, and exiting and entering the substates of cs are recorded in triples (exitcsstnumber, exitcsconsts, exitcscmds), (exitsubstnumber, exitsubconsts, exitsubcmds), and (entersubstnumber, entersubconsts, entersubcmds). The first element of each of these triples is the next available number for the states of the resulting module, such as the numbers associated with the states on lines `#17-#24` in Fig. 25. The second element is a set of constants associated with the states introduced. The third element is a set of commands. The translation of the transitions in cs is recorded in a quadruple (tnsstnumber, tnstrnumber, tnsconsts, tnscmds). Finally, the encodings for the composite substates of cs are, as defined by recursion, recorded in a quadruple (sstrnumber, ssconsts, ssvars, sscmds).

The result of $[\![-]\!]_{\mathcal{S}}$ is a quadruple, whose first element is the next available transition number after the translation of cs and recorded in sstrnumber. The second element of the quadruple is a set of constant declarations including the constant const0 denoting an inactive state, such as the one on line `#35` in Fig. 25, the constants in exitcsconsts, exitsubconsts, entersubconsts, nodesconsts, and ssconsts, and the constants in tnsconsts. The third element of the quadruple is a set of variable declarations including scpc, exit, and the variable declarations in the third element of ssvars. The fourth element of the quadruple is a set of commands including the commands in exitcscmds, exitsubcmds, entersubcmds, tnscmds and sscmds.

The **where** clause in Rule 6 gives more details about the variables in these definitions.

▶ const0 declares a PRISM constant associated with number 0 and representing the inactive status of cs.

▶ scpc declares the program counter variable for cs; its type is a range from 0 to the maximum state number (that is, tnsstnumber − 1). If cs is a machine, the counter is set to a state in PRISM corresponding to the initial junction i0 (that is, a node n of cs that is initial, n ∈ Initial) identified by id(i0), the name of i0 in PRISM. Otherwise, cs is a composite state and inactive initially. So the counter is set to const0.

▶ exit is the declaration of the exit variable for cs.

▶ (exitcsstnumber, exitcsconsts, exitcscmds) is a triple containing a state number, a set of constants, and a set of commands to record the exit of cs. The first element exitcsstnumber is the first state number available. Examples include the numbers on lines `#17-#24` in Fig. 25, for the state machine, and on lines `#27-#31`, for Move. If cs is a state machine (cs ∈ StateMachineDef), there is no need for it to exit, and, therefore, the state number is 1 to record the use of 0 for const0, making 1 the next available number for states, and the sets of constants and commands are empty (∅). Otherwise, the triple is defined by an application of the function exitCompState (omitted here) , which is used to generate the two commands of the module on lines `#52-#54` of Fig. 25. More constants and commands might result if cs has an exit action. Its third argument is the name of scpc (that is, id(cs)`_scpc`). Its last argument 1 is the next available state number.

▶ (exitsubstnumber, exitsubconsts, exitsubcmds) is also a triple defined by the application of the function exitSubstates (whose definition is omitted here). This function is concerned with the encoding of the exit of all substates (that is, all the subnodes s of cs.nodes of cs that are States). The fifth argument of the application is the next available state number as recorded in exitcsstnumber. The first element exitsubstnumber is the new available number. The second element exitsubconsts and the third element exitsubcmds are sets of constants and commands encoding the exit of these substates. This function is used to generate the commands on lines `#26-#28` and the related constants on lines `#9-#10` in Fig. 23 for exiting the state s0 in Fig. 22, and the commands on lines `#48-#51` in Fig. 25 for exiting the state Move in Fig. 17.

▶ (entersubstnumber, entersubconsts, entersubcmds) is a triple defined by applying enterSubstates (omitted) to a set of states that have an entry action (that is, whose action getEntryAction(s) is not null). The fourth argument of the application is the next available state number, exitsubstnumber. The first element entersubstnumber is the new state number, and entersubconsts and entersubcmds are sets of constants and commands used to encode the entering of these substates. This function is used for example, to generate the command on line `#46` in Fig. 25 for entering the simple state Stuck in Fig. 17.

- (nodesstnumber, nodesconsts) is a pair defined by the application of constantsOfNamedElems to all subnodes cs.nodes of cs and to the current available state number (entersubstnumber). This function characterises a set of constants (in nodesconsts) representing states in PRISM that correspond to subnodes of cs like `i0`, `taskState`, and `p0` in the module in Fig. 20. The first element nodesstnumber is the new state number.
- (tnsstnumber, tnstrnumber, tnsconsts, tnscmds) is a quadruple of the next available state and transition numbers, a set of constants, and a set of commands resulting from the translation of the transitions from the nodes cs.nodes of cs by the application of $[\![-]\!]_{\mathcal{TNS}}$ (see Sect. 5.4.3). All commands of `taskSTM` in Fig. 20 are generated by this function.
- (sstrnumber, ssconsts, ssvars, sscmds) is a quadruple resulting from applying the function $[\![-]\!]_{\mathcal{SS}}$ (omitted) to the set of composite states of cs, that is, each node s that is a State and is composite (isComposite(s)). The fifth argument representing the available transition number is from tnstrnumber. The function $[\![-]\!]_{\mathcal{SS}}$ applies $[\![-]\!]_{\mathcal{S}}$ to each composite state recursively.

Next, we discuss how to translate transitions from RoboChart to PRISM and present the rule to do that.

5.4.3 Transitions

As mentioned previously, transitions are translated by $[\![-]\!]_{\mathcal{TNS}}$, whose definition is omitted here. This simple function takes a set of nodes and defines the cumulative result of the application of a function $[\![-]\!]_{\mathcal{TN}}$, which applies to a node n and translates the transitions from this node. We present Rule 7, defining $[\![-]\!]_{\mathcal{TN}}$ if n is a probabilistic junction (ProbJunc), and omit the definition of defining $[\![-]\!]_{\mathcal{TN}}$ for other node types. The result of translating the transitions from p1 in Fig. 26 using $[\![-]\!]_{\mathcal{TN}}$ gives rise to the constant declarations and commands on lines `#3-#8` and `#17-#24` in Fig. 27.

Rule 7. Transitions from a probabilistic junction

$$\left[\!\!\left[\begin{array}{l}\mathsf{n} : \mathsf{ProbJunc}, \mathsf{cs} : \mathsf{NodeContainer}, \mathsf{stm} : \mathsf{StateMachineDef}, \mathsf{exit} : \mathsf{VarDecl}_{\mathsf{pr}},\\ \mathsf{scpcname} : \mathsf{Name}, \mathsf{pcconstrs} : \mathbb{P}\ \mathsf{BoolExpr}_{\mathsf{pr}}, \mathsf{stnumber} : \mathsf{int}, \mathsf{trnumber} : \mathsf{int}\end{array}\right]\!\!\right]_{\mathcal{TN}} : \mathsf{int} \times \mathsf{int} \times \mathbb{P}\ \mathsf{Constant} \times \mathbb{P}\ \mathsf{Command} =$$

$$\begin{array}{l}
\quad (\mathsf{tsstnumber}, \mathsf{trnumber}, \mathsf{tsconsts}, \{\mathsf{cmd}\} \cup \mathsf{tscmds})\\
\mathbf{where}\\
\quad \mathsf{trans} = \{\mathsf{t} : \mathsf{cs.transitions} \mid \mathsf{t.source} = \mathsf{n}\}\\
\quad (\mathsf{tsstnumber}, \mathsf{tsconsts}, \mathsf{tsupdates}, \mathsf{tscmds}) = [\![\mathsf{trans}, \mathsf{n}, \mathsf{stm}, \mathsf{exit}, \mathsf{scpcname}, \mathsf{pcconstrs}, \mathsf{stnumber}, \mathsf{trnumber}]\!]_{\mathcal{TS}}\\
\quad \mathsf{cmd} = [\,]\ \Big(\mathsf{andExprs}(\mathsf{pcconstrs})\ \ \&\ \big(\mathsf{scpcname} = \mathsf{id}(\mathsf{n})\big)\Big) \rightarrow \mathsf{tsupdates};
\end{array}$$

The function $[\![-]\!]_{\mathcal{TN}}$ has eight parameters: n is the node that identifies the transitions to be translated, namely, those with n as a source; cs is the parent of n, either a state machine or a state, of class NodeContainer; stm is the state machine containing n; exit is the declaration of the `exit` variable for cs; scpcname is the name of the program counter variable for cs; pcconstrs is a set of PRISM boolean expressions on the counter variables to identify the control of stm at cs; stnumber is the next available number for a state; and trnumber is the next available number for a transition.

In Rule 7, the result of translating transitions from n is defined in the **where** clause as a quadruple (tsstnumber, tsconsts, tsupdates, tscmds). The first element tsstnumber is the new next available state number; tsconsts is a set of constants corresponding to the extra states for the actions of the transitions, such as those on lines `#3-#8` in Fig. 27; tsupdates is a set of updates that capture the probabilities of the outgoing transitions, such as the three updates on line `#18`; and tscmds is a set of commands for the actions of the outgoing transitions, such as the six commands on lines `#19-#24`. The first and third elements of the result of $[\![-]\!]_{\mathcal{TN}}$ are just tsstnumber and tsconsts. The second element is the argument trnumber, relevant for other forms of nodes, not the probabilistic junctions considered in Rule 7. The fourth element of the result includes those in tscmds and a command cmd encoding the probabilistic choice, such as that on lines `#17-#18` in Fig. 27. The definitions of tsstnumber, tsconsts, tsupdates, tscmds and cmd are in the **where** clause and explained below.

- (tsstnumber, tsconsts, tsupdates, tscmds) results from applying $[\![-]\!]_{\mathcal{TS}}$ to the actual set trans of transitions of cs that are from n. This function defines the cumulative result of applying $[\![-]\!]_{\mathsf{T}}$ (Rule 8), which translates each transition in trans. The commands on lines `#19-#20` and the constants on lines `#3-#4` in Fig. 27 are specified by $[\![-]\!]_{\mathsf{T}}$ when applied to the transition from p1 to s2 in Fig. 26. Similarly, the commands on lines `#21-#22` and `#23-#24`, and the constants on lines `#5-#8` are the result of applying $[\![-]\!]_{\mathsf{T}}$ to the transitions from p1 to s4 and s1. The function $[\![-]\!]_{\mathsf{T}}$ also gives rise to an update: for our example, the three updates on line `#18` for the three transitions from p1 to s1, s2, and s4.

▶ cmd is a command in PRISM encoding the choice made at n probabilistically. Its guard ensures that the counter variables indicate that the current state is that identified by the node n. We have a conjunction of two terms. The first, andExprs(pcconstrs), is itself a conjunction of the boolean expressions in the argument pcconstrs, which are equalities regarding the counter variables for all states and machine containing n. The second conjunct requires that the counter variable for the state or machine containing n, whose name is given by the argument scpcname, has the value id(n) for the probabilistic junction identified by n. The updates of cmd are those in the set tsupdates.

with an entry action, the encoding of t.action leads to the start state of that entry action, represented by the constant named id(t.target)`_entering`, such as `Move_s2_entering` in Fig. 27. Otherwise, target is just the identifier id(t.target) of t.target in PRISM.

▶ targetassigns is a set of assignments to encode entering the target of t when it is a state without entry action. If it is a composite state, we need to update the counter variable id(t.target)`_scpc` of that state to the constant representing its initial junction getInitial(t.target). If the target is a simple state, t is completed by entering its target, and so the lock variable id(stm)`_lock` is set to be free id(stm)`_LOCK_FREE`.

Rule 8. Transition from a probabilistic junction

$$\left[\!\!\left[\begin{array}{l}\mathsf{t : Transition, n : ProbJunc, stm : StateMachineDef, exit : VarDecl_{pr},}\\ \mathsf{scpcname : Name, pcconstrs : \mathbb{P}\ BoolExpr_{pr}, stnumber : int, trnumber : int}\end{array}\right]\!\!\right]_{\mathcal{T}}$$
$$\mathsf{: int \times int \times \mathbb{P}\ Constant \times \mathbb{P}\ Update \times \mathbb{P}\ Command =}$$

$$\mathsf{(astnumber, trnumber, aconsts, \{[\![t.probability]\!]_{e} : aassigns\}, acmds\ )}$$
$$\begin{array}{l}\textbf{where}\\ \quad \mathsf{target} = \textbf{if}\ \ \mathsf{t.target \in State \wedge getEntryAction(t.target) \neq null}\ \textbf{then}\ \ \mathsf{id(t.target)}\texttt{\_entering}\ \ \textbf{else}\ \mathsf{id(t.target)}\\ \quad \mathsf{targetassigns} = \left(\begin{array}{l}\textbf{if}\ \ \mathsf{t.target \in State \wedge getEntryAction(t.target) = null}\ \textbf{then}\\ \quad \textbf{if}\ \ \mathsf{isComposite(t.target)}\ \textbf{then}\ \{\mathsf{id(t.target)}\texttt{\_scpc}' = \mathsf{id(getInitial(t.target))}\}\\ \quad \textbf{else}\ \{\mathsf{id(stm)}\texttt{\_lock}' = \mathsf{id(stm)}\texttt{\_LOCK\_FREE}\}\\ \textbf{else}\ \varnothing\end{array}\right)\\ \quad \mathsf{(astnumber, aconsts, aassigns, acmds)} =\\ \qquad\quad [\![\,\mathsf{t.action, pcconstrs, stnumber, scpcname, null, \{scpcname' = target\} \cup targetassigns}\,]\!]_{\mathcal{A}}\end{array}$$

Rule 8 defines $[\![-]\!]_T$, which is concerned with the translation of a transition t, when its source n is a probabilistic junction. Since t has no trigger or guard, its translation is the result (astnumber, aconsts, aassigns, acmds) of encoding its action t.action and the entering in its target t.target.

Precisely, the result of $[\![-]\!]_T$ is a quintuple, whose first element is the next available state number after the translation of t as recorded in astnumber. The second element is the next available transition number: the argument trnumber. The third and fifth elements are sets of constants and commands resulting from the translation of t.action. The fourth element is a set containing only one update whose probability is the translation of the probability t.probability of t by $[\![-]\!]$e, and whose assignments are the set of assignments resulting from the translation of t.action that updates the counter to the start state for the encoding of the action.

The quadruple (astnumber, aconsts, aassigns, acmds) is defined in the **where** clause described below.

▶ target is the name of the constant representing the final state in PRISM for the encoding of the action t.action. If the target t.target of t is a state

▶ (astnumber, aconsts, aassigns, acmds) is a quadruple defined by application of $[\![-]\!]_{\mathcal{A}}$ to the action t.action of t. The fifth argument null of this function application means the start state of the encoding of t.action is not specified, and so its translation generates a fresh name for the constant for that state. For the action of the transition from p1 to s2 in Fig. 26, the constant to encode its start state in PRISM is `Move_p1_1`. The sixth argument is a set of PRISM assignments containing an update of the counter scpcname for the state or machine of the action to target and targetassigns.

Next, we discuss how to translate actions in RoboChart to PRISM and the definition of $[\![-]\!]_{\mathcal{A}}$.

5.4.4 Actions

Previously in Fig. 20, we show that the action of t2 in Fig. 21 is modelled by five commands shown on lines `#37-#41`, and the action of t3 is modelled by two commands shown on lines `#44-#45`. Generally, an action in RoboChart is modelled by a set of commands with a set of corresponding constants to represent extra states required to encode state change in PRISM.

We define $\llbracket - \rrbracket_{\mathcal{A}}$ to translate a RoboChart action to PRISM. An action act contains a single statement act.action, and so $\llbracket - \rrbracket_{\mathcal{A}}$ is defined by its translation by $\llbracket - \rrbracket_{\mathcal{ST}}$. We define $\llbracket - \rrbracket_{\mathcal{ST}}$ for synchronisation and input or output events in Rule 15 in "Appendix A".

The complete set of translation rules, covering all forms of action, is available in [1]. Next, we present RoboTool, which implements our translation.

6 Automatic model generation

RoboTool[5] supports modelling, validation, and automatic generation of mathematical definitions of RoboChart models written in CSP for use of FDR. We describe here our work extending RoboTool to generate PRISM models automatically: a set of Eclipse plugins[6] that provide textual and graphical modelling tools using the Eclipse Modeling framework (EMF).[7]

We have extended RoboTool's validation facilities with checks of the well-formedness conditions in Sect. 3.3. Fig. 29 shows as an example the implementation for the condition PJ3. The code uses Xtend,[8] a dialect of Java. The method junctionWFC_PJ3 has a probabilistic junction `j` as parameter. It iterates over all transitions in the `parent` (machine or composite state) of `j`, whose source node is `j`. These are all outgoing transitions of `j`. The loop adds the probabilities of these transitions into an array. Afterwards, the method `sumExprEq1` is called to evaluate whether the sum is 1. If not (`sumExprEq1` returns false), RoboTool displays a warning. This check works for probabilities defined by a number, but not more general expressions. In general, checking PJ3 requires theorem proving.

```
@Check
def junctionWFC_PJ3(ProbabilisticJunction j)
{
  val parent = j.eContainer as NodeContainer
  val lstExpr = new ArrayList<Expression>()

  for (t : parent.transitions.filter[t|t.source
        === j]) {
    lstExpr.add(t.probability)
  }

  if(!sumExprEq1(lstExpr)) {
    warning(
        "Sum of probabilities of outgoing
             transitions from a
             ProbabilisticJunction in " + parent
             .name + " might not be equal to 1",
        RoboChartPackage.Literals.
             NAMED_ELEMENT__NAME,
        "probsfromProbJuncNotEqual1"
    )
  }
}
```

Fig. 29 Example: validation of well-formedness conditions

The *RoboChart PRISM Generator* plugin implements the rules from Sect. 5. The implementation of the rules is the way in which models are generated automatically. The implementation uses Epsilon,[9] which provides a collection of languages and tools to support tool development. The first stage of the translation, normalisation, uses the Epsilon Object Language (EOL)[10] for an in-place modification of the RoboChart model. The second stage, transformation, is based on the Epsilon Transformation Language (ETL),[11] a rule-based model-to-model transformation notation.

The transformation of a RoboChart model produces a model in the PRISM metamodel. To generate the textual model accepted by PRISM from the metamodel, we use the Epsilon Generation Language (EGL),[12] a *template-based* language tailored for model-to-text transformation. In the next section, we describe RoboTool facilities for verification using the PRISM model generated automatically.

7 Verification support

PRISM verifies properties defined in probabilistic temporal logics. To improve usability, especially for those without experience with PRISM, we have defined and implemented a property language, called RoboCert. We can specify both properties in CSP (then verify these properties against the standard RoboChart model using FDR) and *probabilistic* properties in temporal logics (LTL, CTL and PCTL), then verify them against the generated PRISM model using PRISM. In Sect. 7.1, we present examples for the mail delivery robot. The syntax of constructs for the probabilistic properties is summarised in Sect. 7.2. The RoboTool implementation is described in Sect. 7.3.

[5] www.cs.york.ac.uk/robostar/robotool/.

[6] robo-star.github.io/robotool-info/.

[7] www.eclipse.org/modeling/emf/.

[8] www.eclipse.org/xtend.

[9] www.eclipse.org/epsilon/.

[10] www.eclipse.org/epsilon/doc/eol/.

[11] www.eclipse.org/epsilon/doc/etl/.

[12] www.eclipse.org/epsilon/doc/egl/.

7.1 Constructs and examples

We present below examples of RoboCert statements, in the context of the module deliverMOD in Fig. 2.

A probabilistic property contains a probabilistic formula with an optional constant configuration to assign the constants that are either declared in the property file or from the RoboChart model with specific values.

Example 2 (Deadlock)

```
constants C1:
  deliverMOD::rp_ref0::batteryCapacity set
    to 20,
  and deliverMOD::rp_ref0::chargeStep set
    to 4
prob property P_deadlock:
  Not Exists [Finally "deadlock"]
  with constants C1
```

Example 2 defines a constants configuration `C1` that sets the constants `batteryCapacity` and `chargeStep` of the robotic platform (reference) `rp_ref0` in `deliverMOD` to 20 and 4. Qualified names uniquely identify an element in the RoboChart model. The property `P_deadlock` specifies that there does not exist a path along which the system deadlocks if the constants are set using `C1`.

We can also define multiple constant configurations for a property, illustrated by Example 3. This feature is useful for design-space exploration (DSE) by analysing multiple designs simultaneously.

Example 3 (Stuck)

```
const x: core::int
constants C2:
  deliverMOD::rp_ref0::batteryCapacity set
    to 20,
  deliverMOD::rp_ref0::chargeStep set to 4,
  and x from set 1:8:1
prob property P_stuck_loc:
  Prob=? [Finally deliverMOD::rp_ref0::p=x&
    deliverMOD::rp_ref0::c=0]
  with constants C2
```

In Example 3, we declare an integer constant `x` and then use it in the property `P_stuck_loc`, which quantitatively measures the probability (`Prob=?`) for the robot to run out of power (`c=0`) in `p` identified by `x`, when `batteryCapacity` and `chargeStep` are set to 20 and 4, and `x` ranges from 1 to 8 (where `1:8:1` denotes a set of integers that is between 1 to 8 inclusive).

RoboCert also supports labels and formulas. A label is a boolean expression, while a formula can be any expression. Example 4 illustrates the usage of labels.

Example 4 (Always get stuck?)

```
label l_stuck =
  deliverMOD::ctrl_ref0::stm_ref0 is in
    deliverMOD::ctrl_ref0::stm_ref0::Stuck
prob property A_stuck:
  Forall [Finally "l_stuck"]
  with constant C1
```

This property states that the robot always finally gets stuck. Here we define a label named `l_stuck`, which is a boolean expression that asserts that movingSTM referenced by `stm_ref0` is at the Stuck state.

Probabilistic properties can also use a reward operator (`Reward=?`). Example 5 demonstrates its usage.

Example 5 (Average number of moves before running out of power)

```
label l_batterystate =
      deliverMOD::ctrl_ref0::stm_ref1 is in
        deliverMOD::ctrl_ref0::stm_ref1::
    batteryState
rewards nbmove =
  [deliverMOD::ctrl_ref0::stm_ref0::move]
    true:1;
endrewards
prob property R_stuck_move:
  Reward {nbmove}=? [
    Reachable {deliverMOD::rp_ref0::c=0}
      &"l_batterystate"]
  with constant C1
```

The reward `nbmove` assigns 1 to each synchronisation over the `move` event from the machine referenced by `stm_ref0` . In the property, the reward operator uses `nbmove` to state the average number of synchronisations over `move` when the robot runs out of power.

Both probability and reward properties support statistical model checking in addition to probabilistic model checking. Statistical model checking uses sample-based discrete-event simulation as in Example 6.

Example 6 (Simulation)

```
prob property P_stuck_loc:
  Prob=? [Finally deliverMOD::rp_ref0::p=x
    & deliverMOD::rp_ref0::c=0]
  using sim with CI at alpha=0.01, n=2000,
    and pathlen=1000
  with constant C1
```

This example applies statistic model checking (`sim`) to verify the property using the `CI` (Confidence Interval) method with supplied parameters. Other methods are `ACI` (Asymptotic Confidence Interval), `APMC` (Approximate Probabilistic Model Checking), and `SPRT` (Sequential Probability

```
1  ProbStatements          ::=  ProbStatement*
2  ProbStatement           ::=  ConstDecl | Constants | Label | Formula | Rewards | ProbProperty
3  ConstDecl               ::=  'const' N ':' Type
4  Constants               ::=  'constants' N ':' ConstConfigs
5  ConstConfigs            ::=  ConstConfig (','ConstConfig)*  ((',')? 'and' ConstConfig)?
6  ConstConfig             ::=  QualifiedNameToElement
7                               (('set' 'to' | 'assigned' | 'with' 'value') Expr
8                                | 'from' 'set' ConstSetExpr)
9  ConstSetExpr            ::=  Expr ':' Expr ':' Expr | '{' (Expr (',' Expr)*)? '}'
10 Label                   ::=  'label' N '=' BoolExpr
11 Formula                 ::=  'formula' N '=' Expr
12 Rewards                 ::=  'rewards' N '=' Reward+ 'endrewards'
13 Reward                  ::=  ('[' QualifiedNameToElement ']')? BoolExpr ':' Expr ';'
14 ProbProperty            ::=  'prob' 'property' N ':' ProbFormula
15                               ('with' ('constant'| 'constants') (ConstConfigs | N))?
16                               ('with' 'cmdoptions' STRING)?
17 ProbFormula             ::=  BoolExpr | StateFormula
18                           |  StateFormula '& & ' StateFormula | StateFormula '||' StateFormula
19                           |  StateFormula '==>' StateFormula | StateFormula '<==>' StateFormula
20 StateFormula            ::= '"' N '"' | 'Not' StateFormula | '(' StateFormula ')' | '{' ProbFormula '}'
21                           | 'Prob' (Bound|Query)? '[' PathFormula ']' (UseMethod)?
22                           | 'Reward' ('{' N '}')? (Bound|Query)? '[' RPathFormula ']' (UseMethod)?
23                           | 'Forall' '[' PathFormula ']' | 'Exists' '[' PathFormula ']'
24 PathFormula             ::= 'Next' ProbOrPathFormula
25                           | ProbOrPathFormula 'Until' (Bound)? ProbOrPathFormula
26                           | 'Globally' (Bound)? ProbOrPathFormula
27                           | 'Weak' 'Until' (Bound)? ProbOrPathFormula
28                           | 'Release' (Bound)? ProbOrPathFormula
29 ProbOrPathFormula       ::= ProbFormula | '[' PathFormula ']'
30 RPathFormula            ::= 'Reachable' ProbFormula | 'Cumul' ExProbOrPathFormula
31                           | 'Finally' (Bound)? ProbOrPathFormula
32                           | 'Globally' (Bound)? ProbOrPathFormula
33                           | 'Weak' 'Until' (Bound)? ProbOrPathFormula
34                           | 'Release' (Bound)? ProbOrPathFormula
35 ProbOrPathFormula       ::= ProbFormula | '[' PathFormula ']'
36 RPathFormula            ::= 'Reachable' ProbFormula | 'Cumul' Expr | 'Total' | 'Instant' Expr
37 Bound                   ::= ('>' | '>=' | '<' | '<=') Expr
38 Query                   ::= '=' '?' | 'min' '=' '?' | 'max' '=' '?'
39 UseMethod               ::= 'using' 'sim' 'with' SimMethod (',' 'and' 'pathlen' '=' Expr)?
40 QualifiedNameToElement  ::= NamedElement ('::' NamedElement)*
```

Fig. 30 BNF syntax of the probabilistic constructs in the property language (`::=` - a metasymbol and interpreted as 'is defined as'; | - separation of alternatives; `Name` - a terminal or non-terminal symbol 'Name'; `?` - appear zero or one time; `+` - repeat one or more times; `*` - repeat any number of times; `'const'` - keyword 'const'; `( )` - group one or more symbols; `N` - a special terminal `ID` for identifiers;)

Ratio Test). Information about these methods can be found online.[13]

7.2 Syntax

The BNF syntax of the probabilistic constructs of RoboCert is sketched in Fig. 30.

There are six types of statements (`ProbStatement`) to specify probabilistic properties: constant declarations (`ConstDecl`), and configurations (`Constants`), label (`Label`), formula (`Formula`), and reward definitions (`Rewards`), and probabilistic properties (`ProbProperty`).

A `ConstDecl` associates a constant name (`N`) to a RoboChart type (`Type`). We reuse `Type` in RoboChart and omit its rule here. The constants that are either declared in the property file or from the RoboChart model are configured via constructs defined on lines #4-#9 in Fig. 30. A configuration name is given and various constants (identified by `QualifiedNameToElement`) are set to specified values (`Expr`). `QualifiedNameToElement` on line #40 uniquely identifies a RoboChart model element through fully qualified names. In the configuration rule on line #8, the value of a constant can be chosen from a set (`ConstSetExpr`) that are defined by either an extension such as `{1,2}` or an integer interval such as `a:b:c` for integers ranging from `a` to `b` by step `c`.

[13] www.prismmodelchecker.org/manual/RunningPRISM/StatisticalModelChecking.

The production rules for labels, formulas, and rewards as shown on lines #10-#13 are straightforward.

A probabilistic property has a name (N), and a probability formula (ProbFormula) under specific constant configurations (defined by a ConstConfigs or a reference to an existing configuration by N). A property may also have associated command options (STRING), which are passed to the PRISM tool directly.

A probability formula can be a boolean expression (BoolExpr), or a state formula (StateFormula), or a composition of two state formulas by logic conjunction, disjunction, implication, or biconditional. A state formula can refer to a defined label name enclosed within a string (at line #22), or could be a negation of another state formula, or a probability measure with a bound (Bound) or a quantitative measure (Query), or a reward measure, or a CTL universal or existential quantifier over paths. A probability measure is over a path formula (PathFormula), while a reward measure is over a reward path formula (RPathFormula). Path formulas include general LTL operators, but with optional bounded variants, where an additional time bound is imposed on the property being satisfied. Reward path formulas specify different types of reward properties: reachability rewards, cumulative rewards, total rewards, or instantaneous rewards.

Both probability and reward measures support statistical as well as probabilistic model checking. Statistical model checking uses sample-based discrete-event simulation. A state formula can specify the simulation methods to be used (UseMethod), and a simulation method is configured using SimMethod.

7.3 A RoboCert plugin in RoboTool

A plugin is available in RoboTool to generate PRISM properties from RoboCert properties. The plugin accepts a property file, parses it, translates constructors in the file to their counterparts in the PRISM property specification language or the PRISM model language, and finally generates a PRISM property file. The plugin provides a modelling environment to edit properties with syntax and error highlighting. It also has content assist through scoping for qualified names.

Translated formulas and rewards are part of PRISM models, instead of property files. Therefore, we need to update the translated PRISM model. The plugin adds the formula and reward definitions from the properties to the PRISM model, and generates the corresponding PRISM properties. RoboTool then runs multiple instances of the PRISM command line tool to verify the generated properties in parallel.

Upon a successful verification, results are shown in a report. Otherwise, error messages indicate problems in the model or properties. The plugin translates the elements in error messages from PRISM back to RoboChart. This improves traceability of errors and results. Counterexamples found by PRISM can be linked back to the original RoboChart model.

RoboTool has been used in the verification of several examples[14] using both probabilistic and statistical model checking. They show that the translation time does not add burden to the verification: a couple of seconds, for verifications that take minutes.

8 Conclusions and future work

Previous work [2] has shown modelling and verification of functional behaviour using RoboChart and RoboTool. This work covers its extension for probabilistic modelling and verification. We have introduced a new construct, probabilistic junctions, for modelling of probabilistic software systems. This new construct impose extra well-formedness conditions. We have extended RoboTool to support modelling of probabilistic systems, including checking of these conditions.

We have derived a metamodel for PRISM and developed and formalised a probabilistic semantics of RoboChart in PRISM. It covers the most challenging constructs of hierarchical state machines and the component model of RoboChart covering parallel controllers.

We have developed support for automatic verification of probabilistic RoboChart models by translating them to the PRISM notation, and then using the PRISM model checker. The translation is in two steps, normalisation of RoboChart models and transformation of normalised models to PRISM. We have formalised and presented here both steps. The translation is automated in RoboTool, and so PRISM models can be generated from RoboChart models automatically.

We have also extended the standard property language [1, Section 5.1] in RoboTool and developed probabilistic property constructs. We have based this on the PRISM property language, but favour a controlled natural language syntax. We use qualified names for references to RoboChart elements. These names are also used in error logs from model checking. This makes model checking tools transparent to RoboTool users. The property language allows RoboTool to verify each property using one instance of PRISM. For instance, RoboTool uses 10 PRISM instances to verify 10 properties in parallel. This reduces model checking time (to the longest checking time of a property).

The translation presented in this paper cannot deal with all features in RoboChart, and our immediate future work is to extend it in order to overcome current limitations. The extension includes support of time primitives, asynchronous connections, operations defined in controllers, during actions, and richer abstract data types to relax TR-TY1 and corresponding expressions. The most significant restric-

[14] www.cs.york.ac.uk/robostar/case_studies/.

tion is the handling of time primitives. The other restrictions just require an additional encoding. Regarding time, our translation is restricted to models that correspond to the Markov models we use (DTMCs and MDPs). RoboChart has a more sophisticated model of time that complements this basic model, and translation of this richer time model is our future work. Many robotic models, however, do not require the super-dense model of time that results from combining the two time-models. So our work caters for many robotic applications.

For the examples we have considered so far, the transformations take a couple of seconds. We do not expect that it raises issues of scalability. Larger case studies and physical modelling are our future work.

The translation defined in this paper is unidirectional from RoboChart to PRISM. It is also feasible to define a translation from PRISM to RoboChart; this is actually not very challenging when compared to the translation defined here. This is also part of our future work to form a bidirectional translation.

It is possible to verify generated PRISM models using other model checkers that accept the same notation, such as Storm and MRMC. Use of other tools that accept different languages, such as the MODEST Toolset, requires different translations. They can, however, capitalise on the normalisation defined here, since it simplifies the structure of models.

RoboChart supports verification of robotic systems by model checking, but the long-term plan is combined use of model checking and theorem proving to deal with larger models and collections. Our immediate future work is to establish a link between model checking and theorem proving in order to verify probabilistic systems in RoboChart complementary. One scenario is to verify a RoboChart model using both model checking and theorem proving, and then compare their results. Another scenario is to use theorem proving as a guaranteed simplifier to simplify part of a RoboChart model (such as state machines, transitions, and states) in order to reduce the complexity for model checking. This could improve model checking performance dramatically.

To support theorem proving, and to connect RoboChart's CSP and PRISM semantics, we are pursuing a unifying theory of CSP and PRISM. This is based on Hoare and He's Unifying Theories of Programming [4]. In [14], we define the probabilistic semantics of the RoboChart action language in a new theory, and use the weakest completion technique [15] to embed the theory of designs [4,73] (for total correctness) in the probabilistic semantic domain. We call this new theory this theory of probabilistic designs. We are mechanising the theory in Isabelle/UTP. This allows us to analyse non-reactive RoboChart models. Our next step is to develop a reactive probabilistic design semantics. Our proof technique is to use reactive relations and Kleene algebra-based verification of reactive programs [74] to calculate contracts for RoboChart models.

With mechanised reactive probabilistic designs semantics, we have a UTP semantics for DTMCs and MDPs (then for PRISM). With that, given that CSP also has a theory of reactive designs, RoboChart and PRISM will then have the same semantic foundation, and we can establish soundness of our transformation with respect to the CSP work.

Using reactive probabilistic designs, we can calculate contracts for RoboChart. Those cut unnecessary internal states and variables generated during transformation. This reduces the state space size of transformed models and will improve model checking performance.

In addition to RoboChart, we are also developing tools and techniques that make use of RoboChart to generate robotic simulations and tests automatically. After these lines of future work are complete, formal studies on usability of these tools and techniques in the robotics domain can be conducted.

Acknowledgements This work is funded by the EPSRC grants EP/M025756/1 and EP/R025479/1, and by the Royal Academy of Engineering grant CiET1718/45. The icons used in RoboChart have been made by Sarfraz Shoukat, Freepik, Google, Icomoon and Madebyoliver from www.flaticon.com, and are licensed under CC 3.0 BY.

A Translation rules

A.1 Normalisation

The function addLoopStateTrans defined in Rule 9 deals with a machine in which there are states having no outgoing transitions. So that the model does not satisfy NFM-1. This function specifies a new loop state and corresponding transitions from the original states to the loop state.

Rule 9. Normalisation of states without outgoing transitions

$$\mathsf{addLoopStateTrans}\,(\mathsf{stm} : \mathsf{NodeContainer}) : \mathbb{P}\,\mathsf{State} \times \mathbb{P}\,\mathsf{Transition} =$$

$$\begin{array}{l}
\quad \textbf{if}\ \ \exists\, \mathsf{n} : \mathsf{stm.nodes} \bullet \mathsf{n} \in \mathsf{State} \land (\forall\, \mathsf{t} : \mathsf{stm.transitions} \bullet \mathsf{t.source} \neq \mathsf{n})\ \ \textbf{then} \\
\qquad (\{\mathsf{loopstate}\}, \{\mathsf{looptran}\} \cup \mathsf{newtrans}) \\
\quad \textbf{else} \\
\qquad (\varnothing, \varnothing) \\
\textbf{where} \\
\quad \mathsf{loopstate} = \mathsf{newState}\,(\mathsf{uname}(\mathsf{stm}, \text{``loop''})) \\
\quad \mathsf{looptran} = \mathsf{newTransition}\,(\mathsf{uname}(\mathsf{stm}, \text{``loop\_self''}), \mathsf{loopstate}, \mathsf{loopstate}) \\
\quad \mathsf{newtrans} = \bigcup \left\{ \mathsf{n} : \mathsf{stm.nodes} \;\middle|\; \begin{array}{l} \mathsf{n} \in \mathsf{State} \land (\forall\, \mathsf{t} : \mathsf{stm.transitions} \bullet \mathsf{t.source} \neq \mathsf{n}) \\ \quad \bullet \{\mathsf{newTransition}\,(\mathsf{uname}(\mathsf{n}, \text{``to\_loop''}), \mathsf{n}, \mathsf{loopstate})\} \end{array} \right\}
\end{array}$$

The first parameter of the function is a NodeContainer, the parent class of both StateMachineDef and State. So this function is applicable to both state machine definitions such as stm in Rule 1, and composite states. If the NodeContainer has one or more nodes n that have no outgoing transitions $(\forall t : stm.transitions \bullet t.source \neq n)$, then the result is a pair formed by the set $\{loopstate\}$ containing a new state loopstate, and a set of transitions ($\{looptran\} \cup newtrans$) including a self-transition looptrans for loopstate and transitions in newtrans from the states without outgoing transitions to loopstate. Otherwise, there is no need to introduce loop, and the result is a pair of empty sets. The **where** clause of the rule has three definitions explained below.

► loopstate is a new state specified by a function newState with a fresh name specified by uname(stm, "loop"). The function newState(name) characterises a State whose name takes the value name, but whose nodes, transitions, and actions are empty. So, it is a simple state without transitions or actions. We note that it is possible to directly define what newState specifies using a class instantiation $\langle name \leadsto \cdots \rangle_{State}$. We define newState, however, to facilitate reuse and hide other fields of the class State that are set to empty or null for simplicity. The introduction of the function newTransition is similar.

► looptran is a new self-transition for loopstate, defined by a function newTransition with three parameters: the name, the source, and the target of the new transition. Other components of the transitions are null: it has no label.

► newtrans is a set of new transitions. For each state n without any outgoing transitions, this set includes one transition from n to the new state loopstate.

If a transition from a state or a normal junction to a normal junction has no action, the model does not satisfy NFM-2. Such a transition, however, can be combined with outgoing transitions from its target normal junction. We identify these transitions as combinable transitions through a function isCombinableTran defined by Rule 11. In a state machine, all normal junctions that have at least one combinable incoming transition are identified by a function inTransCombinableJuncs defined in Rule 10. Such a normal junction and its incoming and outgoing transitions are then combined by a function combTransJunctions defined in Rule 12. Details of these rules are explained next.

Rule 10. Normal junctions whose incoming transitions are combinable

$$\mathsf{inTransCombinableJuncs}\,(\mathsf{stm} : \mathsf{StateMachineDef}) : \mathbb{P}\,\mathsf{Junction} =$$

$$\{\mathsf{n} : \mathsf{stm.nodes} \mid (\mathsf{n} \in \mathsf{Junction} \land \mathsf{n} \notin \mathsf{Initial} \land \mathsf{n} \notin \mathsf{ProbJunc} \land \exists\, \mathsf{t} : \mathsf{stm.transitions} \bullet \mathsf{t.target} = \mathsf{n} \land \mathsf{isCombinableTran}(\mathsf{t})\ )\}$$

The function $\mathsf{inTransCombinableJuncs}$ takes a state machine as argument, and identifies a subset of its nodes n. They are normal junctions, that is, junctions ($\mathsf{n} \in \mathsf{Junction}$), but not initial ($\mathsf{n} \notin \mathsf{Initial}$) and not probabilistic ($\mathsf{n} \notin \mathsf{ProbJunc}$), and at least one transition t among their incoming transitions ($\mathsf{t.target} = \mathsf{n}$) needs to be combined.

Rule 11. Combinable transition

$\mathsf{isCombinableTran}\,(\mathsf{t} : \mathsf{Transition}) : \mathsf{boolean} =$

$$\mathsf{t.action} = \mathsf{null} \land \mathsf{t.source} \notin \mathsf{Initial} \land \mathsf{t.source} \notin \mathsf{ProbJunc}$$

The function $\mathsf{combTransJunction}$ defined in Rule 12 has two parameters: a normal junction j and the set of its incoming and outgoing transitions trans. If the set $\mathsf{intransother}$ of incoming transitions of j that cannot be combined is not empty, then the result is a triple containing the empty set, because j is not removed, the set $\mathsf{intranscomb}$ of the incoming transitions that can be combined, and do need to be removed, and the set $\mathsf{newtrans}$ of the transitions resulting from the combination. Otherwise, the resulting triple includes $\{\mathsf{j}\}$, so that j is removed, the set trans, so that all transitions to and from j are removed, and the set of new transitions $\mathsf{newtrans}$. All the sets are defined in the **where** clause.

Rule 12. Combination of transitions of a junction

$\mathsf{combTransJunction}\,(\mathsf{j} : \mathsf{Junction}, \mathsf{trans} : \mathbb{P}\,\mathsf{Transition}) : \mathbb{P}\,\mathsf{Junction} \times \mathbb{P}\,\mathsf{Transition} \times \mathbb{P}\,\mathsf{Transition} =$

$$\begin{array}{l}
\quad \mathbf{if}\ \ \mathsf{intransother} \neq \varnothing\ \mathbf{then}\ \ (\varnothing, \mathsf{intranscomb}, \mathsf{newtrans})\ \mathbf{else}\ \ (\{\mathsf{j}\}, \mathsf{trans}, \mathsf{newtrans}) \\
\mathbf{where} \\
\quad \mathsf{intranscomb} = \{\mathsf{t} : \mathsf{trans} \mid \mathsf{t.target} = \mathsf{j} \land \mathsf{isCombinableTran}(\mathsf{t})\} \\
\quad \mathsf{intransother} = \{\mathsf{t} : \mathsf{trans} \mid \mathsf{t.target} = \mathsf{j}\} \setminus \mathsf{intranscomb} \\
\quad \mathsf{outtrans} = \{\mathsf{t} : \mathsf{trans} \mid \mathsf{t.source} = \mathsf{j}\} \\
\quad \mathsf{newtrans} = \left\{ \begin{array}{l} \mathsf{ti} : \mathsf{intranscomb};\ \mathsf{tj} : \mathsf{outtrans} \bullet \\ \quad \left\langle \begin{array}{l} \mathsf{name} \rightsquigarrow \mathsf{uname}(\mathsf{ti}, \mathsf{tj.name}), \mathsf{source} \rightsquigarrow \mathsf{ti.source}, \mathsf{target} \rightsquigarrow \mathsf{tj.target}, \\ \mathsf{trigger} \rightsquigarrow \mathsf{ti.trigger}, \mathsf{condition} \rightsquigarrow \mathsf{ti.condition} \land \mathsf{tj.condition}, \mathsf{action} \rightsquigarrow \mathsf{tj.action} \end{array} \right\rangle_{\mathsf{Transition}} \end{array} \right\}
\end{array}$$

As already mentioned, the need to combine a transition t is characterised by $\mathsf{isCombinableTran}$ defined by Rule 11. Such transition has no action ($\mathsf{t.action} = \mathsf{null}$) and its source node $\mathsf{t.source}$ is not an initial or probabilistic junction.

The function $\mathsf{combTransJunctions}$ has two parameters: juncs is a set of normal junctions whose incoming transitions can be combined with outgoing transitions, and trans is a set of incoming and outgoing transitions of these normal junctions. If juncs is empty, then the result is just a triple of empty sets, because there are no transitions to be combined, and so no junctions or transitions to be removed, and no new transitions. Otherwise, the result is basically the cumulative result of applying the function $\mathsf{combTransJunction}$ to each junction. We omit the definition of $\mathsf{combTransJunctions}$ here, but the complete set of rules can be found in [1].

- $\mathsf{intranscomb}$ is a set of incoming transitions t of j that can be combined (and so removed). This is characterised by $\mathsf{isCombinableTran(t)}$.
- $\mathsf{intransother}$ has all other incoming transitions of j.
- $\mathsf{outtrans}$ is the set of outgoing transitions from j.
- $\mathsf{newtrans}$ is a set of new transitions that combine each incoming transition ti from $\mathsf{intranscomb}$ and each outgoing transition tj from $\mathsf{outtrans}$. The new transition has a unique name $\mathsf{uname(ti, tj.name)}$, the same source as ti, the same target of tj, the trigger of ti, and the action of tj. The condition is the conjunction of that of ti and that of tj.

Rule 13. Transition split $\mathsf{splitTran}\,(\mathsf{t} : \mathsf{Transition}) : \mathbb{P}\ \mathsf{Transition} \times \mathbb{P}\ \mathsf{ProbJunc} =$

$$
\begin{array}{l}
\textbf{if}\ \ \mathsf{t.trigger} \neq \mathsf{null} \wedge \mathsf{t.action} \neq \mathsf{null}\ \textbf{then} \\
\quad \left(\left\{ \begin{array}{l} \left\langle \begin{array}{l} \mathsf{name}{\leadsto}\mathsf{uname}(\mathsf{t}, \text{“sp\_1”}), \mathsf{source}{\leadsto}\mathsf{t.source}, \mathsf{target}{\leadsto}\mathsf{pj}, \\ \mathsf{trigger}{\leadsto}\mathsf{t.trigger}, \mathsf{condition}{\leadsto}\mathsf{t.condition} \end{array} \right\rangle_{\mathsf{Transition}}, \\ \left\langle \begin{array}{l} \mathsf{name}{\leadsto}\mathsf{uname}(\mathsf{t}, \text{“sp\_2”}), \mathsf{source}{\leadsto}\mathsf{pj}, \mathsf{target}{\leadsto}\mathsf{t.target}, \\ \mathsf{probability}{\leadsto}1.0, \mathsf{action} = \mathsf{t.action} \end{array} \right\rangle_{\mathsf{Transition}} \end{array} \right\}, \{\mathsf{pj}\} \right) \\
\textbf{else} \\
\quad (\{\mathsf{t}\}, \varnothing) \\
\textbf{where} \\
\quad \mathsf{pj} = \langle \mathsf{name}{\leadsto}\mathsf{uname}(\mathsf{t}, \text{“sp\_pj”}) \rangle_{\mathsf{ProbJunc}}
\end{array}
$$

If a transition has a trigger and an action, the model does not satisfy NFM-3. We, therefore, define splitTran in Rule 13 to define the result of splitting such a transition into two transitions connected via a new probabilistic junction.

Rule 14. Mapping from operations required by state machines to PRISM actions $\mathsf{op2ActionMaps}\,(\mathsf{stm} : \mathsf{StateMachineDef}) : \mathsf{OperationSig} \leftrightarrow \mathsf{Action} =$

$$
\left\{ \mathsf{op} : \mathsf{getRequiredOperations}(\mathsf{stm}) \bullet \mathsf{op} \mapsto \langle \mathsf{name}{\leadsto}\mathsf{uname}\,(\mathsf{stm}, \mathsf{op.name}) \rangle_{\mathsf{Action_{pr}}} \right\}
$$

Rule 15. Statement (CommunicationStmt)

$$
\left[\!\left[\begin{array}{l} \mathsf{stmt} : \mathsf{CommunicationStmt}, \mathsf{pcconstrs} : \mathbb{P}\ \mathsf{BoolExpr_{pr}}, \mathsf{stnumber} : \mathsf{int}, \mathsf{scpcname} : \mathsf{Name}, \\ \mathsf{curstate} : \mathsf{Constant_{pr}}, \mathsf{assigns} : \mathbb{P}\ \mathsf{Assignment_{pr}} \end{array} \right]\!\right]_{\mathcal{ST}} \\
: \mathsf{int} \times \mathbb{P}\ \mathsf{Constant} \times \mathbb{P}\ \mathsf{Assignment} \times \mathbb{P}\ \mathsf{Command} =
$$

$$
\begin{array}{l}
\quad (\ \textbf{if}\ \ \mathsf{curstate} \neq \mathsf{null}\ \ \textbf{then}\ \ \mathsf{stnumber}\ \textbf{else}\ \ \mathsf{stnumber} + 1, \textbf{if}\ \ \mathsf{curstate} \neq \mathsf{null}\ \ \textbf{then}\ \ \varnothing\ \textbf{else}\ \ \{\mathsf{const}\}, \mathsf{curassgns}, \{\mathsf{cmd}\}\ ) \\
\textbf{where} \\
\quad \mathsf{const} = \textbf{if}\ \mathsf{curstate} \neq \mathsf{null}\ \ \textbf{then}\ \ \mathsf{curstate}\ \textbf{else}\ \texttt{const int}\ \mathsf{uname}\,(\mathsf{stmt}, \mathsf{stnumber}) = \mathsf{stnumber}\ ; \\
\quad \mathsf{assgnchvar} = \left(\begin{array}{l} \textbf{if}\ \ \mathsf{stmt.communication.type} = \mathsf{OUTPUT} \vee \mathsf{stmt.communication.type} = \mathsf{SYNC}\ \textbf{then} \\ \quad \{\, \texttt{EVT\_\_}\mathsf{id}\,(\mathsf{stmt.communication.event})' \ = [\![\mathsf{stmt.communication.value}]\!]_{e}\ \} \\ \textbf{else}\ \varnothing \end{array} \right) \\
\quad \mathsf{curassgns} = \{\, \mathsf{scpcname}' = \mathsf{const.name}\,\} \cup \mathsf{assgnchvar} \\
\quad \mathsf{assgninvar} = \left(\begin{array}{l} \textbf{if}\ \ \mathsf{stmt.communication.type} = \mathsf{INPUT}\ \textbf{then} \\ \quad \{\, \mathsf{id}(\mathsf{stmt.communication.parameter})' \ = \texttt{EVT\_\_}\mathsf{id}\,(\mathsf{stmt.communication.event})\ \} \\ \textbf{else}\ \varnothing \end{array} \right) \\
\quad \mathsf{cmd} = [\mathsf{id}(\mathsf{stmt.communication.event})]\ (\mathsf{andExprs}(\mathsf{pcconstrs})\ \ \&\ (\mathsf{scpcname} = \mathsf{const.name})) \rightarrow \texttt{1.0} : \mathsf{assigns} \cup \mathsf{assgninvar}
\end{array}
$$

The function splitTran defines the normalisation of a transition t with a trigger and an action. In this case, the transition is split into two transitions connected via a new probabilistic junction pj, defined in the **where** clause to have a unique name uname(t, “sp_pj”). The result of splitTran(t) is a pair including the two new transitions, and the new junction {pj}. In the set of transitions, we have a transition to pj, and a transition from pj. The transition to pj has the trigger and the condition of t, and the outgoing transition has the action of t and a probability value 1. If t does not have both a trigger and an action, the result is the set containing t itself, and the empty set, since no new junction is needed.

A.2 Transformation to PRISM

Rule 14 defines the function op2ActionMaps used in Rule 4. It has one parameter: stm of type StateMachineDef, and characterises a relation from operations to PRISM actions. This relation establishes a mapping from each operation op in the operations getRequiredOperations(stm) required by stm to a PRISM action with a fresh name specified by uname(stm, op.name). (We recall that this is used to collect information about how operations are used in each machine, so that they can be encoded in the platform module.) For the previously discussed operation moveSteps in Sect. 5.4.1, the result is {moveSteps $\mapsto$ stm_moveSteps_act}.

Rule 15 defines the function $\llbracket - \rrbracket_{\mathcal{ST}}$ that translates a statement. The parameter stmt is a CommunicationStmt, which contains a Communication encapsulating a synchronisation, an input or an output event. The function $\llbracket - \rrbracket_{\mathcal{ST}}$ also has the following parameters: pcconstrs records constraints on the counters that identify the state in which stmt is executed; stnumber is the next available number for states; scpcname is the name of the current counter; curstate is the optional declaration of a PRISM constant representing the state at the start of stmt; and assigns is a set of PRISM assignments that encode the change to the state after the execution of stmt. The constant curstate, if present (not null), indicates that stmt is an entry action. In this case, it is used by all transitions to the state with that entry action.

The result of $\llbracket - \rrbracket_{\mathcal{ST}}$ is a quadruple. The first element is the new available state number. The second is a set of new constants to identify extra states, of which there may be several if the action is compound. The third element stm is a set curassgns of assignments to update the state of the module to the start state for the encoding of stmt. Finally, the fourth element is a set containing one command cmd to encode stmt. If curstate is not null, the supplied curstate encodes the start of stmt, and so no new constant is needed: the new available state number is still stnumber and the result has an empty set of constants. Otherwise, a fresh constant, such as `taskState_3`, is declared and associated with stnumber as shown by the definition of const in the **where** clause. In this case, the new available state number is stnumber + 1, and the set of constants includes a single element const. For the application of $\llbracket - \rrbracket_{\mathcal{ST}}$ to translate the statement in the action of t1 in Fig. 21, the first element of the result is 7 because 6 identifies `taskState_3` on line `#17` in Fig. 20, the second is the declaration of `taskState_3` on line `#17`, the third is the set containing `scpc'=taskState_3` on line `#33`, and the fourth element is the command on line `#34`. The variables used in Rule 15 are defined in the **where** clause explained below.

- ▶ const is the constant encoding the start of the statement stmt. If the argument curstate is not null, the constant is just curstate. Otherwise, it is a constant with a fresh name given by uname(stmt, stnumber) and associated with stnumber.
- ▶ assgnchvar is a set of assignments. If stmt is an output event (its communication is of type OUTPUT) or a synchronisation (its communication is of type SYNC), it contains an assignment to update the event related variable `EVT__`id(stmt.communication.event) from the value $\llbracket$stmt.communication.value$\rrbracket$e carried by the event, to encode communication. Otherwise, it is empty.
- ▶ curassgns contains an assignment to update the counter variable to the start state for the encoding of stmt and assignments in assgnchvar explained above.
- ▶ assgninvar is also a set of assignments. If stmt is an input event (that is, its communication is of type INPUT), this set contains an assignment to update the input variable id(stmt.communication.parameter) to the value of the variable used for output, to encode the input communication. Otherwise, assgninvar is empty since there is no data exchange.
- ▶ cmd is a command. Its action is the name of the event of stmt given by id(stmt.communication.event). Its guard ensures that the counter variables indicate that the current state is that identified by const. We have a conjunction of two terms. The first, andExprs(pcconstrs), is itself a conjunction of the boolean expressions in the argument pcconstrs, which are equalities regarding the counter variables for all states and state machine containing stmt. The second conjunct requires that the counter variable for the state or machine containing stmt, whose name is given by the argument scpcname, has the value const.name for the encoding of stmt. The update of cmd has probability `1` and assignments containing the argument assigns to update the counters to the final state of the encoding of stmt and assgninvar explained above.

References

1. Miyazawa, A., Cavalcanti, A., Ribeiro, P., Li, W., Woodcock, J., Timmis, J.: RoboChart Reference Manual. University of York, Tech. rep. (2018) www.cs.york.ac.uk/circus/publications/techreports/reports/robochart-reference.pdf
2. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., Timmis, J., Woodcock, J.: RoboChart: modelling and verification of the functional behaviour of robotic applications. Softw. Syst. Model. (2019). https://doi.org/10.1007/s10270-018-00710-z
3. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV, *Lecture Notes in Computer Science*, vol. 6806, pp. 585–591. Springer (2011) dblp.uni-trier.de/db/conf/cav/cav2011.html#KwiatkowskaNP11
4. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall (1998)
5. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall Int. (1985)
6. Roscoe, A.W.: Understanding Concurrent Systems. Texts in Computer Science. Springer (2011)
7. Cavalcanti, A.L.C., Woodcock, J.C.P.: A Tutorial Introduction to CSP in Unifying Theories of Programming. In: Refinement Techniques in Software Engineering. Lecture Notes in Computer Science, vol. 3167, pp. 220–268. Springer-Verlag (2006). https://doi.org/10.1007/11889229_6www-users.cs.york.ac.uk/~alcc/publications/papers/CW06.pdf

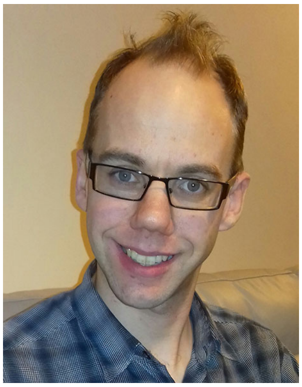
8. Baxter, J., Ribeiro, P., Cavalcanti, A.: Sound reasoning in tock-csp. Acta Informatica (**in press**)
9. FDR: Failures-Divergences Refinement https://www.cs.ox.ac.uk/projects/fdr/
10. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3: A modern refinement checker for CSP. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 187–201 (2014)
11. Goldsmith, M., East, I., Duce, D., Green, M., Martin, J., Welch, P.: CSP: The best concurrent-system description language in the world—probably! Communicating Process Architectures, pp. 227–232 (2004)
12. Foster, S., Zeyda, F., Woodcock, J.C.P.: Isabelle/UTP: A mechanised theory engineering framework. In: UTP 2015, LNCS, vol. 8963, pp. 21–41. Springer (2015)
13. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer (2002)
14. Woodcock, J.C.P., Cavalcanti, A.L.C., Foster, S., Mota, A., Ye, K.: Probabilistic semantics for RoboChart: A weakest completion approach. In: Unifying Theories of Programming. Lecture Notes in Computer Science, p. to appear, Springer (2019)
15. He, J., Morgan, C., McIver, A.: Deriving probabilistic semantics via the 'weakest completion'. In: Davies, J., Schulte, W., Barnett, M. (eds.) Formal Methods and Software Engineering, pp. 131–145. Springer, Berlin Heidelberg, Berlin, Heidelberg (2004)
16. Conserva Filho, M.S., Marinho, R., Mota, A., Woodcock, J.: Analysing RoboChart with probabilities. Formal Methods: Foundations and Applications (2018). https://doi.org/10.1007/978-3-030-03044-5_13
17. Segala, R., Lynch, N.A.: Probabilistic simulations for probabilistic processes. Nord. J. Comput. **2**(2), 250–273 (1995)
18. Hansson, H.: Time and probabilities in formal design of distributed systems. Department of Computer Systems, Uppsala University, Phd thesis (1991)
19. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Inf. Comput. **94**(1), 1–28 (1991). https://doi.org/10.1016/0890-5401(91)90030-6
20. Stoelinga, M.: An introduction to probabilistic automata. Bull. EATCS **78**, 176–198 (2002)
21. van Glabbeek, R.J., Smolka, S.A., Steffen, B., Tofts, C.M.N.: Reactive, generative, and stratified models of probabilistic processes. In: Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4–7, 1990, pp. 130–141. IEEE Computer Society (1990). https://doi.org/10.1109/LICS.1990.113740
22. Kemeny, J.G., Snell, J.L.: Finite Markov Chains: With a New Appendix "Generalization of a Fundamental Matrix" (Undergraduate Texts in Mathematics). Springer (1983)
23. Kemeny, J.G., Snell, J.L., Knapp, A.W.: Denumerable Markov Chains (1976). https://doi.org/10.1007/978-1-4684-9455-6
24. Howard, R.: Dynamic Probabilistic Systems: Semi-Markov and decision processes. Series in Decision and Control. Wiley (1971) https://books.google.co.uk/books?id=vuZQAAAAMAAJ
25. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming, 1st edn. Wiley (1994)
26. Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A storm is coming: A modern probabilistic model checker. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification, pp. 592–600. Springer, Cham (2017)
27. Alur, R., Henzinger, T.A.: Reactive modules. Formal Methods Syst. Design **15**(1), 7–48 (1999)
28. Bohnenkamp, H., d'Argenio, P.R., Hermanns, H., Katoen, J.P.: Modest: A compositional modeling formalism for hard and softly timed systems. IEEE Trans. Softw. Eng. **32**(10), 812–830 (2006)
29. Hahn, E.M., Hartmanns, A., Hermanns, H., Katoen, J.P.: A compositional modelling and analysis framework for stochastic hybrid systems. Formal Methods Syst. Design **43**(2), 191–232 (2013)
30. Park, H.W., Ramezani, A., Grizzle, J.W.: A finite-state machine for accommodating unexpected large ground-height variations in bipedal robot walking. IEEE Trans. Robot. **29**(2), 331–345 (2013)
31. Rabbath, C.A.: A finite-state machine for collaborative airlift with a formation of unmanned air vehicles. J. Intell. Robot. Syst. **70**(1), 233–253 (2013)
32. Tomic, T., Schmid, K., Lutz, P., Domel, A., Kassecker, M., Mair, E., Grixa, I.L., Ruess, F., Suppa, M., Burschka, D.: Toward a fully autonomous UAV: research platform for indoor and outdoor urban search and rescue. IEEE Robot. Autom. Mag. **19**(3), 46–56 (2012)
33. Liu, W., Winfield, A.F., Sa, J.: Modelling swarm robotic systems: a case study in collective foraging. Towards Auton. Robot. Syst. 25–32 (2007)
34. Konur, S., Dixon, C., Fisher, M.: Analysing robot swarm behaviour via probabilistic model checking. Robot. Auton. Syst. **60**(2), 199–213 (2012)
35. Colvin, R., Grunske, L., Winter, K.: Probabilistic timed behavior trees. In: International Conference on Integrated Formal Methods, pp. 156–175. Springer (2007)
36. Dromey, R.G.: From requirements to design: Formalizing the key steps. In: First International Conference onSoftware Engineering and Formal Methods, 2003. Proceedings., pp. 2–11. IEEE (2003)
37. Beauquier, D.: On probabilistic timed automata. Theor. Comput. Sci. **292**(1), 65–84 (2003)
38. Object Management Group: OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1 (2011) www.omg.org/spec/UML/2.4.1
39. Addouche, N., Antoine, C., Montmain, J.: Uml models for dependability analysis of real-time systems. In: 2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No. 04CH37583), vol. 6, pp. 5209–5214. IEEE (2004)
40. Addouche, N., Antoine, C., Montmain, J.: Combining extended UML models and formal methods to analyze real-time systems. In: International Conference on Computer Safety, Reliability, and Security, pp. 24–36. Springer (2005)
41. Jansen, D.N., Hermanns, H., Katoen, J.: A probabilistic extension of UML statecharts. In: Damm, W., Olderog, E. (eds.) FTRTFT 2002: 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, Co-sponsored by IFIP WG 2.2, Oldenburg, 9–12 September 2002, *Lecture Notes in Computer Science*, vol. 2469, pp. 355–374. Springer (2002)
42. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Asp. Comput. **6**(5), 512–535 (1994). https://doi.org/10.1007/BF01211866
43. Schlegel, C., Hassler, T., Lotz, A., Steck, A.: Robotic soft. systems: from code-driven to model-driven designs. In: ICAR 2009, pp. 1–8. IEEE (2009)
44. Nordmann, A., Hochgeschwender, N., Wigand, D., Wrede, S.: A survey on domain-specific modeling and languages in robotics. In: FTRTFT 2002: 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, Co-sponsored by IFIP WG 2.2, Oldenburg, 9–12 September 2002, vol. 7, pp. 75–99 (2016)
45. Schlegel, C., Worz, R.: The software framework smartsoft for implementing sensorimotor systems. In: Proceedings 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human and Environment Friendly Robots with High Intelligence and Emotional Quotients (Cat. No. 99CH36289), vol. 3, pp. 1610–1616. IEEE (1999)
46. Alonso, D., Vicente-Chicote, C., Ortiz, F., Pastorfranco, J.: V3cmm: a 3-view component meta-model for model-driven robotic software development. J. Softw. Eng. Robot. **1**, 3–17 (2010)

47. Bruyninckx, H., Klotzbücher, M., Hochgeschwender, N., Kraetzschmar, G., Gherardi, L., Brugali, D.: The brics component model: a model-based development paradigm for complex robotics software systems. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, pp. 1758–1764 (2013)
48. Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., Ziane, M.: RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications. In: SIMPAR 2012, pp. 149–160. Springer (2012)
49. Ramaswamy, A., Monsuez, B., Tapus, A.: Saferobots: A model-driven framework for developing robotic systems. In: Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems, pp. 1517–1524 (2014). https://doi.org/10.1109/IROS.2014.6942757
50. Hochgeschwender, N., Gherardi, L., Shakhirmardanov, A., Kraetzschmar, G.K., Brugali, D., Bruyninckx, H.: A model-based approach to software deployment in robotics. In: IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, pp. 3907–3914 (2013)
51. Mallet, A., Pasteur, C., Herrb, M., Lemaignan, S., Ingrand, F.: Genom3: Building middleware-independent robotic components. IEEE Int. Conf. Robot. Automat. , 4627–4632 (2010)
52. Foughali, M., Berthomieu, B., Zilio, S.D., Ingrand, F., Mallet, A.: Model Checking Real-Time Properties on the Functional Layer of Autonomous Robots. In: Formal Methods and Soft. Eng., pp. 383–399. Springer (2016)
53. Berthomieu, B., Vernadat, F.: Time petri nets analysis with TINA. In: Third International Conference on the Quantitative Evaluation of Systems, pp. 123–124. IEEE Computer Society, Riverside, California, USA (2006). https://doi.org/10.1109/QEST.2006.56
54. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 11-15 September 2006, Pune, India, pp. 3–12. IEEE Computer Society (2006). https://doi.org/10.1109/SEFM.2006.27
55. Thrun, S., Burgard, W., Fox, D.: Probabilistic Robotics (Intelligent Robotics and Autonomous Agents). The MIT Press (2005)
56. PRISM Lab Session, Part B: Mail Delivery Robot http://www.prismmodelchecker.org/courses/aims1617/deliveryRobot.php
57. Cavalcanti, A., Barnett, W., Baxter, J., Carvalho, G., Conserva Filho, M., Miyazawa, A., Ribeiro, P., Sampaio, A.: In: Cavalcanti, A., Dongol, B., Hierons, R., Timmis, J., Woodcock, J. (eds.) RoboStar technology: a roboticist's toolbox for combined proof, simulation, and testing. Software Engineering for Robotics. Springer International Publishing (2021)
58. Jansen, D.N., Hermanns, H., Katoen, J.P.: A Probabilistic Extension of UML Statecharts. In: Formal Tec. in Real-Time and Fault-Tolerant Syst., *LNCS*, vol. 2469, pp. 355–374. Springer (2002)
59. Nokovic, B., Sekerinski, E.: Verification and code generation for timed transitions in pcharts. In: Proceedings of the 2014 International C* Conference on Computer Science & Software Engineering, p. 3. ACM (2014)
60. Katoen, J.P., Khattri, M., Zapreevt, I.: A Markov reward model checker. In: Second International Conference on the Quantitative Evaluation of Systems (QEST'05), pp. 243–244. IEEE (2005)
61. PRISM Team: The PRISM Language: Semantics (2008) www.prismmodelchecker.org/doc/semantics.pdf
62. Younes, H.L.S., Kwiatkowska, M.Z., Norman, G., Parker, D.: Numerical vs. statistical probabilistic model checking. STTT **8**(3), 216–228 (2006). https://doi.org/10.1007/s10009-005-0187-8
63. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: 10th Joint Meeting of the European Soft. Eng. Conf. and the ACM SIGSOFT Symp. on the Foundations of Soft. Eng., pp. 263–272. ACM (2005)
64. PRISM Team: The PRISM Language: Reward-based Properties www.prismmodelchecker.org/manual/PropertySpecification/Reward-basedProperties
65. Jensen, H.: Model checking probabilistic real time systems. In: Proc. 7th Nordic Workshop on Programming Theory, pp. 247–261. Citeseer (1996)
66. Kwiatkowska, M., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. Theor. Comput. Sci. **282**(1), 101–150 (2002)
67. Kwiatkowska, M.Z., Norman, G., Parker, D.: Prism 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV. Lecture Notes in Computer Science, vol. 6806, pp. 585–591. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_47 dblp.uni-trier.de/db/conf/cav/cav2011.html#KwiatkowskaNP11
68. Kattenbelt, M., Kwiatkowska, M., Norman, G., Parker, D.: A game-based abstraction-refinement framework for markov decision processes. Formal Methods Syst. Design **36**(3), 246–280 (2010)
69. Kwiatkowska, M., Norman, G., Parker, D., Sproston, J.: Performance analysis of probabilistic timed automata using digital clocks. Formal Methods Syst. Design **29**(1), 33–78 (2006)
70. Woodcock, J.C.P., Davies, J.: Using Z-Specification, Refinement, and Proof. Prentice-Hall (1996)
71. Cavalcanti, A.L.C., Sampaio, A.C.A., Woodcock, J.C.P.: A Refinement Strategy for Circus. Formal Aspects Comput. **15**(2–3), 146–181 (2003)
72. Spivey, J.M.: The Z Notation: A Reference Manual, 2nd. Prentice-Hall (1992)
73. Woodcock, J.C.P., Cavalcanti, A.L.C.: A tutorial introduction to designs in Unifying Theories of Programming. In: Boiten, E.A., Derrick, J., Smith, G. (eds.) IFM 2004: Integrated Formal Methods, *LNCS*, vol. 2999, pp. 40–66. Springer-Verlag (2004). Invited tutorial
74. Foster, S., Ye, K., Cavalcanti, A., Woodcock, J.: Calculational verification of reactive programs with reactive relations and Kleene Algebra. In: International Conference on Relational and Algebraic Methods in Computer Science, pp. 205–224. Springer (2018)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Kangfeng Ye is a research associate at the University of York, UK. With an industrial background in embedded systems for communication and semiconductor, he developed his research interests in formal modelling and verification of Robotics and Cyber-Physical Systems, the areas he is working on now. In his research, he has developed a new theory to verify Simulink diagrams based on theorem proving, and is developing tools to verify probabilistic systems using both theorem proving and model checking. He gained a Ph.D. in Computer Science at the University of York in 2017, and his thesis addresses model checking of state-rich formalisms.

Ana Cavalcanti (University of York) is Professor of Software Verification and Royal Academy of Engineering Chair in Emerging Technologies working on Software Engineering for Robotics: modelling, validation, simulation, and testing. She currently leads the RoboStar research group at the University of York. She held a Royal Society-Wolfson Research Merit Award and a Royal Society Industry Fellowship to work with QinetiQ in avionics. She has chaired the Programme Committee of various well-established international conferences, is on the editorial board of four international journals, and is Chair of the board of the Formal Methods Europe Association. Her current research is on theory and practice of verification and testing for robotics.



Simon Foster is a UKRI Innovation Fellow leading the CyPhyAssure project at the University of York, which explores assurance techniques for cyber-physical systems. His research applies theorem proving to formal verification, and he leads the development of Isabelle/UTP, a practical theorem prover for heterogeneous systems. He has applied Isabelle/UTP to verification tools for process algebras, reactive programs, and hybrid systems. He has also developed Isabelle/SACM, an interactive tool for assurance cases with evidence coming from multiple formal verification tools in Isabelle. As part of CyPhyAssure, he is collaborating with industry to bring these techniques to the real world. He gained his Ph.D. at the University of Sheffield in 2010, which developed a timed process algebra for Web service composition semantics.

Alvaro Miyazawa is a research associate at the University of York. Having completed B.Sc. in Computer Science at the University of Sao Paulo and doctoral research at the University of York, his main research interests are in formal semantics and refinement for domain specific languages and graphical notations, and the development of refinement strategies to support high levels of automation in program verification. Currently, his research focuses on modelling, testing, simulation and verification for robotics.

Jim Woodcock is Professor of Software Engineering at the University of York and Professor of Digital Twins at Aarhus University. His research interests are in the unification of mathematical theories for the cost-effective design of hardware and software components in innovative, safe, and secure cyber-physical systems, including robotics. His scientific work has enabled him to make significant contributions to the application of mathematical techniques in industry in domains of strategic importance to society. He is one of the leaders of the Verifiability research node for the UK Trusted Autonomous Systems Programme. He is a Fellow of the Royal Academy of Engineering.