Language Support for Verifying Reconfigurable Interacting Systems

Downloaded from: https://research.chalmers.se, 2026-09-14 23:22 UTC

Citation for the original published paper (version of record):

Alrahman, Y., Azzopardi, S., Di Stefano, L. et al (2023). Language Support for Verifying Reconfigurable Interacting Systems. International Journal on Software Tools for Technology Transfer, 25(5-6): 765-784. http://dx.doi.org/10.1007/s10009-023-00729-8

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

International Journal on Software Tools for Technology Transfer (2023) 25:765–784
https://doi.org/10.1007/s10009-023-00729-8

FOUNDATIONS FOR MASTERING CHANGE

Special Section: Rigorous Engineering of Collective Adaptive Systems

Language support for verifying reconfigurable interacting systems

Yehia Abd Alrahman[1] · Shaun Azzopardi[1] · Luca Di Stefano[1] · Nir Piterman[1]

Accepted: 10 October 2023 / Published online: 2 November 2023
© The Author(s) 2023

Abstract

Reconfigurable interacting systems consist of a set of autonomous agents, with integrated interaction capabilities that feature opportunistic interaction. Agents seemingly reconfigure their interaction interfaces by forming collectives and interact based on mutual interests. Finding ways to design and analyse the behaviour of these systems is a vigorously pursued research goal. In this article, we provide a modelling and analysis environment for the design of such system. Our tool offers simulation and verification to facilitate native reasoning about the domain concepts of such systems. We present our tool named R-CHECK (please find the associated toolkit repository here: https://github.com/dsynma/recipe). R-CHECK supports a high-level input language with matching enumerative and symbolic semantics and provides modelling convenience for features such as reconfiguration, coalition formation, and self-organisation. For analysis, users can simulate the designed system and explore arising traces. Our included model checker permits reasoning about interaction protocols and joint missions.

Keywords Model checking · Agent theories and models · Verification of multi-agent systems

1 Introduction

Reconfigurable multi-agent systems (MASs) [1, 5] have emerged as new computational systems, consisting of a set of autonomous agents that interact based on mutual interest, thus creating a sort of opportunistic interaction. That is, agents seemingly reconfigure their interaction interfaces and dynamically form groups/collectives based on run-time changes in their execution context. Designing these systems and reasoning about their behaviour is very challenging, due to the high level of dynamism that reconfigurable MASs exhibit. Reconfigurable MASs can be viewed as a special case of collective-adaptive systems (CASs) [38]. Indeed, the only major difference is that the latter focus more on scalable systems with a large number of components. Reconfigurable MASs, instead, focus on small-scale systems with collective behaviour. Reconfigurable MASs are useful for applications in control systems, smart factories, autonomous vehicles, etc., while CASs can be used to explain systems with a very large number of (relatively small) agents. For instance, theories about CASs can be used to reason about the spread of infectious diseases [34], the utilisation of bike-sharing systems [32], etc. That being said, theories on reconfigurable MASs focus more on qualitative measures, while CAS theories are quantitative. In this article, we focus on qualitative analysis (through, e.g., model checking) of reconfigurable systems.

Traditionally, model checking [17, 31] is considered as a mainstream verification tool for reactive systems [9] in the community. A system is usually represented by a low-level language such as NuSMV [16], reactive modules [10, 26], concurrent game structures [11], or interpreted systems [23]. The modelling primitives of these languages are very close to their underlying semantics, e.g., predicate representation, transition systems, etc. Thus, it makes it hard to model and reason about high-level features of reconfigurable MASs such as reconfiguration, group formation, and self-organisation. Indeed, encoding these features in existing formalisms would not only make it hard to reason about them, but will also create exponentially large and detailed models that are not amenable to verification. The latter is a classical challenge for model checking and is often termed as *state-space explosion*.

✉ Y.A. Alrahman
yehia.abd.alrahman@gu.se

S. Azzopardi
shaun.azzopardi@gu.se

L. Di Stefano
luca.di.stefano@gu.se

N. Piterman
nir.piterman@gu.se

[1] University of Gothenburg, Gothenburg, Sweden

Existing techniques that attempt to tame the state-space explosion problem (such as binary decision diagrams [BDDs], abstraction, and bounded model checking [BMC]) can only act as a mitigation strategy, but cannot provide the right level of abstraction to compactly model and reason about high-level features of reconfigurable MASs.

MASs are often programmed using high-level languages that support domain-specific features of MASs like emergent behaviour [3, 4, 37], interactions [5], intentions [18], and knowledge [23]. These descriptions are very involved to be directly encoded in plain transition systems. Thus, we often want programming abstractions that focus on the domain concepts, abstract away from low-level details, and consequently reduce the size of the model under consideration. The rationale is that reasoning about a system requires having the right level of abstraction to represent its behaviour. Thus, there is a pressing demand to extend traditional model-checking tools with support for reasoning about high-level features of reconfigurable MASs. This suggests supporting an intuitive description of programs, actions, protocols, reconfiguration, self-organisation, etc.

ReCiPe [1, 5] is a promising framework to support modelling and verification of reconfigurable MASs. It is supported with both an enumerative semantics and a symbolic semantics and model representation that permits the use of symbolic representation to enable efficient analysis. However, writing programs in ReCiPe is very hard and error-prone. This is because ReCiPe models are encoded in a predicate-based representation that is far from how people usually write programs. In fact, the predicate representation of ReCiPe supports no programming primitives to control the structure of programs, and thus everything is encoded using state variables.

In this article, we present R-CHECK, a toolkit for designing, simulating, and verifying reconfigurable MASs. R-CHECK supports a minimalistic high-level programming language with symbolic semantics based on the ReCiPe framework. The syntax of the language was presented briefly, along with a short case study, in [7].

Here we formally present the syntax and semantics of R-CHECK and use it to model and reason about a non-trivial case study from the realm of reconfigurable and self-organising MASs. We have integrated LTOL [1, 5] – a logic specialised for interaction – into R-CHECK, thus allowing native reasoning about selective interaction strategies.

This article is an extended and enhanced version of the paper in [6]. There are two major new contributions [6]:

(i) We have integrated native reasoning about message exchange by supporting LTOL [1] specifications, an extension to LTL with native support for message exchange. We have devised a new algorithm for LTOL model checking that is different from the one presented in [1] to allow integration with NUXMV. Indeed, rather than a bespoke automaton construction [1], we extend the underlying system with additional information and model check an extended LTL formula over it. This alternative algorithm is integrated into R-CHECK.

(ii) We also now support a native interpreter for the semantics of R-CHECK models, and thus we can now simulate and visualise the counterexamples from the model-checking algorithm directly on the generated symbolic automata. In [6], we could only enable simulation by completely relying on NUXMV, and we had no way to replay model-checking counterexamples. Note that the counterexamples that NUXMV supports are state-based and due to possible non-determinism it was hard to link them to message exchange. Our interpreter instead focuses on message exchange.

This specialised integration provides a powerful and native tool that permits verifying high-level features of reconfigurable MASs. Indeed, we can reason about systems from both an individual and a system level. We show how to reason about synchronisations, interaction protocols, and joint missions and how to express high-level features such as channel mobility, reconfiguration, coalition formation, and self-organisation.

The remainder of this article is structured as follows. In Sect. 2, give a background on ReCiPe [1, 5], the underlying theory of R-CHECK. In Sect. 3, we present the language of R-CHECK and its symbolic semantics. In Sect. 4, we provide a non-trivial case study to model autonomous resource allocation. In Sect. 5, we present the LTOL logic and motivate its use through a case study. In Sect. 6, we present our new algorithm of LTOL embedding in NUXMV. We also discuss the integration of R-CHECK in NUXMV. Finally, we provide concluding remarks in Sect. 7.

2 ReCiPe: a model of computation

We present the underlying theory of R-CHECK and explain its semantics. R-CHECK accepts a high-level language that is based on the symbolic ReCiPe formalism [1, 5]. We briefly present ReCiPe agents and their composition to generate a system-level behaviour. All these definitions are based on *discrete systems* [36].

ReCiPe relies on (attributed) channel communication, where agents agree on channel names to exchange messages. These messages carry data (in variables D) specified by senders. Agents can also specify the target of communication by attributing the channels through predicates, similar to [3, 4]. As opposed to the latter, ReCiPe supports a dynamic reconfiguration of channel utilisation. Moreover, ReCiPe supports two kinds of communication, *channelled*

broadcast and *channelled multicast*. In channelled broadcast, the communication is non-blocking, that is, the communication can still go through if a targeted receiver is not ready to engage. Contrarily, in multicast, the communication is blocked until all targeted receivers are willing to accept the message and engage in the communication. Agents agree on a set of channels CH, which includes the one used exclusively for broadcast, $\star$.

Usually, broadcast is used for service discovery, for instance, when agents are unaware of the existence of each other and want to be discovered or to establish links for further interaction. On the other hand, multicast can capture a more structured interaction where agents have dedicated links to interact on. The reconfiguration of interaction interfaces in ReCiPe makes it possible to integrate the two ways of communication in a meaningful way. That is, agents may start with a flat communication structure and use broadcast to discover others. Thanks to ReCiPe's channel passing feature, agents can dynamically build dedicated communication structures based on channel references they exchange during execution.

In order to target a subset of agents, in an interaction, sending agents rely on *property identifiers*, that is, identifiers that senders use to specify properties required by targeted receivers. The set of property identifiers is PV. For instance, agent k may specify that it wants to communicate on channel "a" with all agents that listen to "a" and satisfy the property "BatteryLevel $\geq 30\%$". In other words, property identifiers PV are used by agents to indirectly specify constraints on the targeted receivers.

Each agent has a way to relate property identifiers to its local state through a relabelling function. As we will see later, we have generalised this function in R-CHECK to deal with more sophisticated expressions. Thus, agents specify properties anonymously using these identifiers, which are later translated to the corresponding receiver's local state. Messages are then only delivered to receivers that satisfy the property after relabelling.

Formally, an agent is defined as a discrete system (DS) [36].

Definition 1 (Agent)
An agent is a tuple $A = \langle V,\ f,\ g^s,\ g^r,\ \mathcal{T}^s,\ \mathcal{T}^r,\ \theta\rangle$, where:

- V is a finite set of typed local variables;
- $f : \text{PV} \to V$ is a function associating property identifiers with local variables;
- $g^s(V, \text{CH}, \text{D}, \text{PV})$ is a send guard specifying the property of the targeted receivers based on the current evaluation of V, CH, and D, which is checked against every receiver j after applying f_j;
- $g^r(V, \text{CH})$ is a receive guard describing the connectedness of an agent to a channel ch; we let $g^r(V, \star) = \mathsf{true}$, i.e., every agent is always connected to the broadcast channel;
- $\mathcal{T}^s(V, V', \text{D}, \text{CH})$ and $\mathcal{T}^r(V, V', \text{D}, \text{CH})$ are assertions describing the send and receive transition relations, respectively; we assume that an agent is broadcast input-enabled, i.e., $\forall v, \mathbf{d}\ \exists v'$ **s.t.** $\mathcal{T}^r(v, v', \mathbf{d}, \star)$;
- θ is an assertion on V describing the initialization of the agent.

In this definition, a state of an agent s is an assignment to the agent's local variables V, i.e., for $v \in V$ if $\mathsf{Dom}(v)$ is the domain of v, then s is an element in $\prod_{v\in V} \mathsf{Dom}(v)$. In the case that all variables range over a finite domain, the number of states is finite. A state is initial if its assignment to V satisfies θ. Note that A is a DS, and thus we use the set V' to denote the primed copy of V. That is, V' stores the next assignment to V. Moreover, we use Id to denote the assertion $\bigwedge_{v\in V} v = v'$. That is, V is kept unchanged. We use $\mathbf{d}$ to denote an assignment to the data variables D. We also abuse the notation and use f for the assertion $\bigwedge_{pv\in \text{PV}} pv = f(pv)$.

Agents exchange messages of the form $m = (ch, \mathbf{d}, i, \pi)$. A message is defined by the channel it is sent on ch, the data it carries $\mathbf{d}$, the sender identity i (we assume a unique identifier for each agent), and the assertion specifying the property of targeted receivers π. The predicate π is obtained by grounding the sender's send guard on the sender's current state, used channel ch, and exchanged data $\mathbf{d}$.

Send transition relations $\mathcal{T}^s$ characterise what messages may be sent, with one message sent at each point in time, while receive transition relations $\mathcal{T}^r$ characterise the reaction of a receiving agent to a message.

We use $\text{KEEP}(X)$ to denote that a set of variables X is not changed by a transition (either send or receive). That is, $\text{KEEP}(X)$ is equivalent to the assertion $\bigwedge_{x\in X} x = x'$.

A set of agents agreeing on property identifiers PV, data variables D, and channels CH defines a *system*. We give the semantics of systems in terms of predicates to facilitate efficient symbolic analysis (through BDD or satisfiability modulo theory SMT). We use $\biguplus$ for disjoint union.

Formally, a ReCiPe system is a DS and is defined as follows.

Definition 2 (System)
Given a set $\{A_i\}_i$ of agents, a system is $S = \langle \mathcal{V},\ \rho,\ \theta\rangle$, where $\mathcal{V} = \biguplus_i V_i$, a state of the system s is in $\prod_i \prod_{v\in V_i} \mathsf{Dom}(v)$, and the initial assertion $\theta = \bigwedge_i \theta_i$. The transition relation ρ of S

is as follows:

$$\rho = \exists ch.\, \exists \textsc{d}.\ \bigvee_k \mathcal{T}_k^s(V_k, V_k', \textsc{d}, ch) \wedge \bigwedge_{j \neq k} \left(\exists \textsc{pv}.f_j \wedge \begin{pmatrix} g_j^r(V_j, ch) \wedge g_k^s(V_k, ch, \textsc{d}, \textsc{pv}) \wedge \mathcal{T}_j^r(V_j, V_j', \textsc{d}, ch) \\ \vee \quad \neg g_j^r(V_j, ch) \wedge \mathsf{Id}_j \\ \vee\ \neg g_k^s(V_k, ch, \textsc{d}, \textsc{pv}) \wedge ch = \star \wedge \mathsf{Id}_j \end{pmatrix} \right)$$

The transition relation ρ describes two modes of interactions: blocking multicast and non-blocking broadcast. Formally, ρ relates a system state s to its successors s' given a message $m = (ch, \mathbf{d}, k, \pi)$. Namely, there exists an agent k that sends a message with data $\mathbf{d}$ (an assignment to $\textsc{d}$) with assertion π (an assignment to g_k^s) on channel ch, and all other agents (a) are connected to channel ch, satisfy the send predicate π, and participate in the interaction (i.e., have a corresponding receive transition for the message), (b) are not connected and idle, or (c) do not satisfy the send predicate of a broadcast and idle. That is, the agents satisfying π (translated to their local state by the conjunct $\exists \textsc{pv}.f_j$) and connected to channel ch (i.e., $g_j^r(s^j, ch)$) get the message and perform a receive transition. As a result of the interaction, the state variables of the sender and these receivers might be updated. The agents that are *not connected* to the channel (i.e., $\neg g_j^r(s^j, ch)$) do not participate in the interaction and stay still. In case of broadcast, namely, when sending on $\star$, agents are always connected and the set of receivers not satisfying π (translated again as above) stay still. Thus, a blocking multicast arises when a sender is blocked until all *connected* agents satisfy $\exists \textsc{pv}.f_j \wedge \pi$. The relation ensures that when sending on a channel different from $\star$, the set of receivers is the full set of *connected* agents. On the broadcast channel agents not satisfying the send predicate do not block the sender.

Example: Consider a ReCiPe system that is composed of two agents A_1 and A_2, agreeing on the set of channels $\textsc{ch} = \{\star\}$, the data variables $\textsc{d} = \{\textsc{msg}, \textsc{lnk}\}$, and the property variables $\textsc{pv} = \{pv\}$. Here, we use non-boolean variables to simplify the presentation.

A_1 is defined as follows:
- $V_1 = \{\mathsf{cLink} : \mathtt{channel},\ \mathsf{role} : \mathtt{enum}\}$
- $f_1 = \{pv \mapsto \mathsf{role}\}$
- g_1^s is $(ch = \star \wedge pv = \mathsf{client})$
- g_1^r is true
- $\mathcal{T}_1^s$ is $(\textsc{keep}(\mathsf{V}_1) \wedge \mathbf{d}(\textsc{msg} \mapsto \mathsf{join}, \textsc{lnk} \mapsto \mathsf{c}) \wedge \mathsf{ch} = \star)$
- $\mathcal{T}_1^r$ is $\textsc{keep}(V_1)$
- θ_1 is $(\mathsf{cLink} = \mathsf{c} \wedge \mathsf{role} = \mathsf{client})$

That is, A_1 has two local variables, cLink of channel type and role of enumerate type. Moreover, A_1 relabels the property identifier pv locally as the value of its local variable role. The send predicate g_1^s indicates that A_1 intends to interact on the broadcast channel $\star$ with agents that satisfy the property $pv = \mathsf{client}$ according to their local relabelling. The receive predicate g_1^r indicates that A_1 is always enabled to receive.

Behaviour-wise, A_1 can send a message join with a link c on the broadcast channel $\star$. Moreover, A_1 is not willing to receive any messages.

Initially, the local variables of A_1 are set such that cLink is assigned link c and role is a client.

A_2 is defined as follows:
- $V_2 = \{\mathsf{cLink} : \mathtt{channel},\ \mathsf{role} : \mathtt{enum}\}$
- $f_2 = \{pv \mapsto \mathsf{role}\}$
- g_2^s is false
- g_2^r is true
- $\mathcal{T}_2^s$ is false
- $\mathcal{T}_2^r$ is $(\mathsf{cLink} = \bot \wedge \mathsf{cLink}' = \mathbf{d}(\textsc{lnk}) \wedge \textsc{keep}(\mathsf{role}) \wedge \mathbf{d}(\textsc{msg} \mapsto \mathsf{reserve}) \wedge \mathsf{ch} = \star)$
- θ_2 is $(\mathsf{cLink} = \bot \wedge \mathsf{role} = \mathsf{client})$

Clearly, A_2 only differs from A_1 with respect to the send guard, the send transition relation (which are set to false), the receive transition relation (which indicates that A_2 is willing to receive a message named reserve and stores the value of $\textsc{lnk}$ of the message in cLink), and the initial condition where cLink is set to $\bot$.

By applying Definition 2, we see that the composition of A_1 and A_2 indeed forms a ReCiPe system (where local variables of A_1 and A_2 are joined with disjoint union to account for similar local naming).

Now starting from the initial conditions of both agents, we apply the system transition relation ρ. Clearly, there exist only one message that satisfies ρ, namely, the message on channel $\star$, and data variables assigned as $\{\textsc{msg} \mapsto \mathsf{join}, \textsc{lnk} \mapsto \mathsf{c}\}$, where A_1 is the sender (i.e., its send transition relation $\mathcal{T}_1^s$ is satisfied). Moreover, there is only one receiver A_2 which is connected to $\star$ (i.e., g_2^r is satisfied), its receive transition relation $\mathcal{T}_2^r$ is satisfied with respect to the same message, and the send guard g_1^s is $(ch = \star \wedge pv = \mathsf{client})$ in conjunction with local relabelling of A_2 (i.e., $pv = \mathsf{role}$) is satisfiable. Thus, ρ holds and as a result A_2 sets its local cLink variable to c that is communicated in the message. In the next cycle, ρ is checked again based on the new updated states.

Clearly, ReCiPe is a low-level formalism that is geared towards efficient BDD representation and model checking; thus, it is not meant to be used as a modelling language. R-CHECK, on the other hand, builds on the efficient representation of ReCiPe and provides a set of high-level primitives that can be used for modelling purposes.

R-CHECK adopts a symbolic model-checking approach that directly works on the predicate representation of ReCiPe systems. Technically speaking, the behaviour of each ReCiPe agent is represented by a first order predicate that is defined as a disjunction over the send and receive transition relations

```
1   channel: identifier, ···, identifier
2   message−structure: var_name:type, ···, var_name:type
3   property−variables: var_name:type, ···, var_name:type
4   enum name {identifier, ···, identifier}
    ⋮
5   ⋮
6   enum name {identifier, ···, identifier}
```

Fig. 1 R-CHECK script prelude

```
1   agent name
2     local:
3       var_name:type, ···, var_name:type
4
5     init: θ_T
6
7     relabel:
8       predicate_var <− Exp
              ⋮
9             ⋮
10      predicate_var <− Exp
11
12        receive−guard: g^r(V_T, CH)
13    repeat: P
```

Fig. 2 An agent type

of that agent. Moreover, both send and receive transition relations can be represented by a disjunctive normal form predicate of the form $\bigvee(\bigwedge_j \mathsf{assertion}_j)$, that is, a disjunct of all possible send/receive transitions enabled in each step of a computation. In the following, we will define a high-level language that can be used to write user-friendly programs with symbolic computation steps. We will also show how to translate these programs to the ReCiPe predicate representation.

3 The R-CHECK language

We formally present the syntax of R-CHECK and show how to translate it to the ReCiPe predicate representation.

An R-CHECK program starts with a prelude, reported in Fig. 1, explicitly defining the communication context: by defining the channel names (line 1), the data variables a message carries (line 2), and the property variables (line 3). Moreover, the user is allowed to define custom enum types (lines 4–6).

After the communication context is defined, the user can define the set of agents that compose the systems. We define agent behaviour as data types.

We now introduce the `agent` type, its structure, and how to instantiate it; we also introduce the syntax of the agent behaviour and how to create a system of agents.

The type `agent` is reported in Fig. 2. Intuitively, each agent type has a `name` that identifies a specific type of behaviour. As we will see later, we permit creating multiple instances/copies with the same type of behaviour. Each agent has a local state `local` represented by a set of local variables V_T, each of which can be of a boolean, integer, or enum type. The initialisation of an agent `init:` θ_T is a predicate characterising the initial assignments to the agent local variables. The section `relabel` is used to implement the relabelling function of predicate variables in a ReCiPe agent. Here, we allow the relabelling to include a boolean expression `Exp` over local variables V_T to accommodate a more expressive relabelling mechanism, e.g., $pv \leftarrow (\mathsf{length} \geq 20)$. The section `receive-guard` specifies the connectedness of the agent to channels given a current assignment to its local variables. Syntax-wise, to specify receive guards we use a special variable ch to denote the channel a message was sent on at the current time step; thus, we can write $\mathsf{ch} = a$ to denote that an agent is always receptive to messages sent on channel a. Moreover, an agent is always receptive to messages on broadcast, i.e., $\mathsf{ch} = *$ is implicitly added as a disjunct to whatever receive guard the user writes. The latter permits input-enabled broadcast as explained in Definition 1. The non-terminating behaviour of an agent is represented by `repeat: P`, which executes the process `P` indefinitely.

Before we introduce the syntax of agent behaviour, we show how to instantiate an agent and how to compose the different agents to create a system.

An agent type of name A can be instantiated as follows: $A(id,\theta)$. That is, we create an instance of A with identity id and an additional initial restriction θ. Here, we take the conjunction of θ with the predicate θ_T in the `init` section of type A as the initial condition of this instance.

We use the parallel composition operator $\|$ to inductively define a system as shown in the following production rule.

$$\text{(System)} \qquad S ::= \quad A(id,\theta) \quad | \quad S_1 \| S_2$$

That is, a system is either an instance of agent type or a parallel composition of a set of instances of (possibly) different types. The semantics of $\|$ is fully captured by ρ in Definition 2.[1]

The syntax of an R-CHECK process is inductively defined as follows.

$$\begin{array}{lll} \text{(Process)} & P ::= & C;P \quad | \quad P+P \quad | \quad \mathsf{rep}\ P \quad | \quad C \\ \text{(Command)} & C ::= & l:C \quad | \quad \langle\Phi\rangle x!\,\pi\ \mathbf{d}\ \mathsf{U} \quad | \quad \langle\Phi\rangle x?\,\mathsf{U} \end{array}$$

An agent behaviour corresponds to an infinite repetition of a process. A process P is either a command prefix process $C;P$, a non-deterministic choice between two processes $P+P$, a loop $\mathsf{rep}\ P$, or a command C. There are three types of commands corresponding to either a labelled command, a message-send, or a message-receive. A command of the form $l:C$ is a syntactic labelling and is used to allow the model

[1] Technically, in the case that the relabelling uses a predicate, we have to introduce a variable of the correct type and make sure that every transition changing the state of the agent updates this variable to the value of the given predicate, that is, given the relabelling $pv \rightarrow \mathtt{Exp}$, adding a variable pv to local variables and the conjunct $pv' = \mathtt{Exp}'$ to all transitions.

checker to reason about syntactic elements as we will see later.[2] A command of the form $\langle\Phi\rangle x!\ \pi\ \mathbf{d}\ \mathsf{U}$ corresponds to a message-send. Intuitively, the predicate Φ is an assertion over the current assignments to local variables, i.e., it is a pre-condition that must hold before the transition can be taken; x is a placeholder (or a bound name) for a channel name. Note that x may refer to the value of a local variable, since we allow local variables to have the type `channels`. As the names suggest, π and $\mathbf{d}$ are the sender predicate and the assignment to data variables (i.e., the actual content of the message), respectively. Lastly, U is the next assignment to local variables after taking the transition. We use ! to distinguish send transitions. A command of the form $\langle\Phi\rangle x?\,\mathsf{U}$ corresponds to a message-receive. Differently from message-send, Φ can also predicate on the incoming message, i.e., the assignment $\mathbf{d}$. We use ? to distinguish receive transitions.

Despite the minimalistic syntax of R-CHECK, we can express every control flow structure in a high-level programming language. For instance, by combining non-determinism and pre-conditions of commands, we can encode every structure of an IF statement. Similarly, we can encode finite loops by combining $\mathsf{rep}\ P$ and commands C, e.g., $(\mathsf{rep}\ C1 + C_2)$ means: repeat C_1 or block until C_2 happens.

We define a system by instantiating agent types and put them in parallel, as shown below.

```
1   system = agent_name(Id1, θ1) || ... || agent_name(Id2, θ1)
```

Finally, the user can supply logical specifications/ properties of the behaviour of the system as a set of LTL and LTOL formulas as shown below.

```
1   LTL ltl_spec
2   ...
3   LTL ltl_spec
4   LTOL ltol_spec
5   ...
6   LTOL ltol_spec
```

3.1 The semantics of R-CHECK

We initially give a structural semantics[3] to the R-CHECK process using a finite automaton such that each transition in the automaton corresponds to a symbolic transition. Intuitively, the automaton represents the control structure of an R-CHECK process. We will further use this automaton alongside the agent definition to give an R-CHECK agent an execution semantics based on the symbolic ReCiPe framework. This two-step semantics will help us in verifying structural properties of R-CHECK agents.

Definition 3 (Structure automaton)
A structure automaton is of the form $G = \langle S, \Sigma, s_i, E\rangle$, where:

- S is a finite set of states;
- $s_i \in S$ is the *initial* state;
- Σ is the alphabet of G; and
- $E \subseteq S \times \Sigma \times S$ is the set of edges of G.

We use (s_1, σ, s_2) to denote an edge $e \in E$ such that s_1 is the source state of e, s_2 is the target state of e, and the letter σ is the label of e.

Now, everything is in place to define the structure semantics of R-CHECK processes. We define a function $(\!|\ .\ |\!)^{[s_i,s_f]} : P \to 2^E$, which takes an R-CHECK process P as input and produces the set of edges of the corresponding structure automaton. The function $(\!|\ .\ |\!)^{[s_i,s_f]}$ returns a set of transitions corresponding to the input process, starting from state s_i and (possibly) finishing at state s_f. The definition is reported below.

$$(\!|P_1; P_2|\!)^{[s_i,s_f]} \triangleq (\!|P_1|\!)^{[s_i,s_1]} \bigcup (\!|P_2|\!)^{[s_1,s_f]} \qquad \text{for a fresh } s_1$$

$$(\!|P_1 + P_2|\!)^{[s_i,s_f]} \triangleq (\!|P_1|\!)^{[s_i,s_f]} \bigcup (\!|P_2|\!)^{[s_i,s_f]}$$

$$(\!|\mathsf{rep}\ P|\!)^{[s_i,s_f]} \triangleq (\!|P|\!)^{[s_i,s_i]}$$

$$(\!|C|\!)^{[s_i,s_f]} \triangleq \{(s_i, C, s_f)\}$$

Given a process P appearing in the body of the agent type under repeat, we construct its corresponding structure automaton by constructing the set of edges $E = (\!|P|\!)^{[s_i,s_i]}$, given some state s_i. Let S and Σ be the set of states and commands used in E, respectively. Then the corresponding structure automaton is $\langle S, \Sigma, s_i, E\rangle$.

Note that the states of the structure automaton only represent the control structure of the process, and an agent can have multiple initial states depending on θ_T while starting from s_i.

We explain informally the semantics. $(\!|P_1; P_2|\!)^{[s_i,s_f]}$ is the union of the transitions created by P_1 and P_2 while creating a fresh state in the graph s_1 to allow sequentiality, where P_1 starts in s_i and ends in s_1 and later P_2 continues from s_1 and ends in s_f. That is, the structure of the process is encoded using an extra memory. Differently, the non-deterministic choice $(\!|P_1 + P_2|\!)^{[s_i,s_f]}$ does not require extra memory because the execution of P_1 and P_2 is independent. The semantics of $(\!|\mathsf{rep}\ P|\!)^{[s_i,s_f]}$ is similar to that of $(\!|\mathsf{repeat}: P|\!)^{[s_i,s_f]}$ and is introduced to allow finite looping inside a non-terminating process. Finally, the semantics of a command in C corresponds to an edge $\{(s_i, C, s_f)\}$ in the structure automaton. This means that the alphabet Σ of the automaton ranges over R-CHECK commands. Note that the

[2] This option is made redundant by the support of LTOL. However, it is left to support backward compatibility and for convenience.

[3] We use the term structural semantics instead of symbolic because we want to stress that this semantics exposes the control structure of a process.

translation is completely syntactic and does not enumerate variable values, resulting in a symbolic automaton.

To translate an R-CHECK agent into a ReCiPe agent, we first introduce the functions typeOf, varsOf, predOf, and guardOf on a command C. That is, typeOf(C) returns the type of a command C as either ! or ?. For example, typeOf($\langle\Phi\rangle\ ch\ !\ \pi\ \mathbf{d}\ \mathsf{U}$) = !. Moreover, varsOf($C$) returns the set of local variables that are updated in C, while predOf(C) returns the predicate characterising C in terms of local variables V_T, the primed copy V_T', the channel ch, and the data variables D (excluding π). For instance,

$$\mathsf{predOf}(\langle \mathsf{Link} = \mathsf{c}\rangle\star!\pi(\mathsf{MSG} := \mathsf{m})[\mathsf{Link} := \mathsf{b}])$$

is

$$(\mathsf{Link} = \mathsf{c}) \wedge (\mathsf{ch} = \star) \wedge (\mathsf{MSG} = \mathsf{m}) \wedge (\mathsf{Link}' = \mathsf{b})$$

That is, we provide a predicate that uniquely characterises the information in the command.

Finally guardOf(C) returns the send predicate π in a send command and false otherwise.

Next we define how to construct a ReCiPe agent from an R-CHECK agent with structure semantics interpreted as a structure automaton.

Definition 4 (From R-CHECK to ReCiPe) Given an instance of agent type T as defined in Fig. 2 with a structure semantics interpreted as a structure automaton $G = \langle S, \Sigma, s_i, E\rangle$, we can construct a ReCiPe agent $A = \langle V, f, g^s, g^r, \mathcal{T}^s, \mathcal{T}^r, \theta\rangle$ that implements its behaviour.

We construct A as follows:

- $V = V_T \cup \{\mathsf{st}\}$ is the union of the set of declared variables V_T in the `local` section of T in Fig. 2 and a new state variable st ranging over the states S in G of the structure automaton, representing the control structure of the process of T. Namely, the control structure of the behaviour of T is now encoded as an additional variable in A.
- $\mathcal{T}^s =$
$$\bigvee_{(s_1,\sigma,s_2)\in E\ :\ \mathsf{typeOf}(\sigma)=\ !} \begin{pmatrix} \mathsf{predOf}(\sigma) \wedge (\mathsf{st} = s_1) & \wedge \\ (\mathsf{st}' = s_2) \wedge \textsc{keep}(V_T \backslash \mathsf{varsOf}(\sigma)) & \end{pmatrix}$$
- $\mathcal{T}^r =$
$$\bigvee_{(s_1,\sigma,s_2)\in E\ :\ \mathsf{typeOf}(\sigma)=\ ?} \begin{pmatrix} \mathsf{predOf}(\sigma) \wedge (\mathsf{st} = s_1) & \wedge \\ (\mathsf{st}' = s_2) \wedge \textsc{keep}(V_T \backslash \mathsf{varsOf}(\sigma)) & \end{pmatrix}$$
- $g^s = \bigvee_{\sigma\in\Sigma:\ \mathsf{typeOf}(\sigma)=\ !} \mathsf{guardOf}(\sigma)$
- The initial condition $\theta = \theta_T \wedge (\mathsf{st} = s_i)$ is the conjunction of the initial condition θ_T in the `init` section of T in Fig. 2 and the predicate $\mathsf{st} = s_i$, specifying the initial state of G.
- f and g^r have one-to-one correspondence in section `relabel` and section `receive-guard`, respectively, of T in Fig. 2.

Namely, the structure of the R-CHECK process is encoded as a state variable st in ReCiPe. The send transition relation is encoded as the disjunction of all edges labelled with send commands, and similarly for the receive transition relation. The send guard is a disjunction of all guards in send commands. Lastly, the initial condition of the ReCiPe model is initialised to the initial state of the state variable.

4 Case study: autonomous resource allocation

We model a scenario where a group of clients are requested to jointly solve a problem. Each client will buy a computing virtual machine (VM) from a resource manager and use it to solve their task. Initially, clients know the communication link of the manager, but they need to self-organise and coordinate the use of the link anonymously. The manager will help establishing connections between clients and available machines, and later clients proceed interacting independently with machines on private links learnt when the connection is established.

There are two types of machines: high-performance machines and standard ones. The resource manager commits to providing high-performance VMs to clients, but when all of these machines are reserved, the clients are assigned to standard ones. The protocol proceeds until each client buys a machine, and then all clients have to collaborate to solve the problem and complete the task.

To model this scenario in R-CHECK we need three agent types: client, manager, and machine. Each type can be instantiated multiple times, to model scenarios of different size. We continue by defining each agent type. We assume an enum type that identifies the role of each agent: $\mathsf{rolevals} = \{\mathsf{client}, \mathsf{manager}, \mathsf{machine}\}$.

A client uses the local variables cLink, mLink, and tLink of type channels and role : rolevals to control its behaviour, where cLink is a link to interact with the manager, mLink is a placeholder for a mobile link that can be learnt at run-time, tLink is a link to synchronise with other clients to complete the task, and role is the role of the client. A client's initial condition θ_{c} is

$$\mathsf{cLink} = \mathsf{c} \wedge \mathsf{mLink} = \mathsf{empty} \wedge\ \mathsf{tLink} = \mathsf{t} \wedge \mathsf{role} = \mathsf{client}$$

specifying that the resource manager is reachable on c, the mobile link is empty, the task link is t, and the role is client.

Note that the interfaces of agents are parameterised to their local states and state changes may create dynamic and

opportunistic interactions. For instance, when cLink is set to empty, the client does not connect to channel c; also, when a channel is assigned to mLink, the client starts receiving messages on that channel.

In our example, clients are not aware of the existence of each other while they share the resource manager channel c. Thus, they may coordinate to use the channel anonymously by means of broadcast. A client reserves the channel c by means of a broadcast message with a predicate targeting other clients. All other clients self-organise and disconnect from c and wait for a release message.

A message in R-CHECK carries an assignment to a set of data variables D. In our scenario, $\mathrm{D} = \{\mathrm{LNK}, \mathrm{MSG}\}$, where LNK is used to exchange a link with other agents and MSG denotes the label of the message and takes values from

$$\{\mathsf{reserve}, \mathsf{request}, \mathsf{release}, \mathsf{buy}, \mathsf{connect}, \mathsf{full}, \mathsf{complete}\}$$

Agents in this scenario use one predicate variable pv ranging over roles to specify potential receivers. Remember that every agent i has a relabelling function $f_i : \mathrm{PV} \to V_i$ that is applied to the send guard once a message is delivered to check whether it is eligible to receive. For a client, $f_c(\mathsf{pv}) = \mathsf{role}$. The send guard of a client appears in the messages that the client sends, as we will explain later. In general, broadcasts are destined to agents assigning to the predicate variable pv a value matching the role of the sender, i.e, client; messages on cLink are destined to agents assigning mgr to pv; and other messages are destined to everyone listening on the right channel.

The receive guard g_c^r is

$$(\mathsf{ch} = \star) \vee (\mathsf{ch} = \mathsf{cLink}) \vee (\mathsf{ch} = \mathsf{tLink})$$

That is, reception is always enabled on broadcast and on a channel that matches the value of cLink or tLink. Note that these guards are parameterised to local variables and thus may change at run-time, creating a dynamic communication structure.

The behaviour of the client is reported in Fig. 3. In this example, we label each command with a name identifying the message and its type (i.e., s for send and r for receive). For instance, the send transition at line 2 is labelled with sReserve, while the receive transition at line 4 is labelled with rReserve. We use these later to reason about agent interactions syntactically.

Initially in lines 2–6, every client may either broadcast a reserve message to all other clients (i.e., (pv = role)) or receive a reserve message from one of them. This is to allow clients to self-organise and coordinate to use the common link. That is, a client may initially reserve an interaction session with the resource manager by broadcasting a reserve message to all other clients, asking them to disconnect the

```
1  repeat: (
2          (sReserve: <cLink==c > ∗! (pv==role)(MSG := reserve)[]
3          +
4          rReserve: <cLink==c && MSG == reserve> ∗?
5          [cLink := empty]
6          )
7                ;
8          (
9          sRequest: <cLink!=empty> cLink! (pv==mgr)
10         (MSG := request)[]
11         ;
12          rConnect: <mLink==empty && MSG == connect>
       cLink? [mLink := LNK]
13         ;
14         sRelease: <TRUE> ∗! (pv==role)(MSG := release)
15         [cLink := empty]
16         ;
17         sBuy: <mLink!=empty> mLink! (TRUE)(MSG := buy)
18         [mLink := empty]
19         ;
20         (
21           sSolve: <TRUE> tLink!(TRUE)(MSG := complete)[]
22          +
23           rSolve:<MSG == complete> tLink? []
24         )
25        +
26         rRelease: <cLink==empty && MSG == release> ∗?
27          [cLink := c]
28        )
29       )
```

Fig. 3 Client behaviour

common link c (stored in their local variable cLink), or receive a reserve message, i.e., it gets asked by another client to disconnect from channel c. In either case, the client continues to line 8. Depending on what happened in the previous step, the client may proceed to establish a session with the resource manager (i.e., (pv = mgr)) and a machine (lines 9–25) or gets stuck waiting for a release message from the client, currently holding the session (lines 26–27). In the latter case, the client gets back in the loop to line 1 after receiving a release message and attempts again to establish the session.

In the former case, the client uses the blocking multicast channel c to send a request to the resource manager (line 9) and waits to receive a private connection link with a VM agent on cLink (line 12). When the client receives the connect message on cLink, the client assigns its mLink variable the value of LNK in the message. That is, the client is now ready to communicate on mLink. The agent releases the common link c on line 14 by broadcasting a release message to all other clients (with (pv = role)) and proceeds to line 17 and starts communicating privately with the assigned VM agent. The client buys a service from the VM agent on a dedicated link stored in mLink by sending a buy to the VM agent to complete the transaction. The client proceeds to line 20 and waits for other clients to collaborate and finish the task. Thus, the client either initiates the last step and sends a complete message when the rest of the clients are ready (line 21) or receives a complete message from another client that is ready (line 23).

We now specify the manager and the VM and show how reconfigurable multicast can be used to cleanly model a point-to-point interaction.

```
1   repeat: (
2           rRequest: <MSG == request> cLink? [];
3           sForward: <TRUE> hLink! (TRUE)(MSG := request)[]
4           ;
5           (
6            rConnect: <MSG == connect> cLink? []
7            +
8
9            rep ( rFull: <MSG == full> hLink? [];
10                 sRequest: <TRUE> sLink! (TRUE)
11                 (MSG := request)[]
12           )
13          )
14          )
```

Fig. 4 Manager behaviour

The resource manager's local variables are

$$\mathsf{hLink},\ \mathsf{sLink},\ \mathsf{cLink},\ \mathsf{role}$$

where hLink and sLink store channel names to communicate with high- and standard-performance VMs, respectively, and the rest are as defined before.

The initial condition θ_{m} is

$$\mathsf{hLink} = \mathsf{g}_1 \wedge \mathsf{sLink} = \mathsf{g}_2 \wedge \mathsf{cLink} = \mathsf{c} \wedge \mathsf{role} = \mathsf{mgr}$$

Note that the link g_1 is used to communicate with the group of high-performance machines, while g_2 is used for standard ones.

The send guard for a manager is always satisfied (i.e., g^s_m is true), while the receive guard specifies that a manager only receives on broadcast or on channels that match with cLink or hLink, i.e., g^r_m is

$$(\mathsf{ch} = \star) \vee (\mathsf{ch} = \mathsf{cLink}) \vee (\mathsf{ch} = \mathsf{hLink})$$

The behaviour of the agent manager is reported in Fig. 4. In summary, the manager initially forwards requests received on channel c (line 2) to the high-performance VMs as in line 3. The negotiation protocol with machines is reported in lines 5–13. The manager can receive a connect message and directly enable the client to connect with the VM as in line 6 or receive a full message, informing that all high-performance machines are fully occupied (line 9). In the latter case, the requests are forwarded to the standard-performance machines on sLink as in lines 10–11. The process repeats until a connect message is received (line 6) and the manager gets back to line 1 to handle other requests. Clearly, the specifications of the manager assume that there are plenty of standard VMs but not a lot of high-performance ones. Thus, it only expects a full message to be received on channel hLink. Note also that the manager gets ready to handle the next request once a connect message is received on channel c and leaves the client and the selected VM to interact independently.

The VM's local variables are

$$\mathsf{gLink},\ \mathsf{pLink},\ \mathsf{cLink},\ \mathsf{asgn}$$

```
1   repeat: (
2           rForward: <cLink==empty && MSG == request> gLink?
        [cLink:= c];
3           (
4            sConnect: <cLink==c && !asgn> cLink! (TRUE)(MSG
        := connect, LNK := pLink)[cLink:= empty, asgn:= TRUE]
5            +
6            sFull: <cLink==c && asgn> gLink! (TRUE)(MSG :=
        full)[cLink:= empty]
7            +
8            rConnect: <cLink==c && MSG == connect> cLink?
        [cLink:= empty]
9            +
10           rFull: <cLink==c && asgn && MSG == full> gLink?
        [cLink:= empty]
11          )
12          +
13          rBuy: <MSG == buy> pLink? []
14          )
```

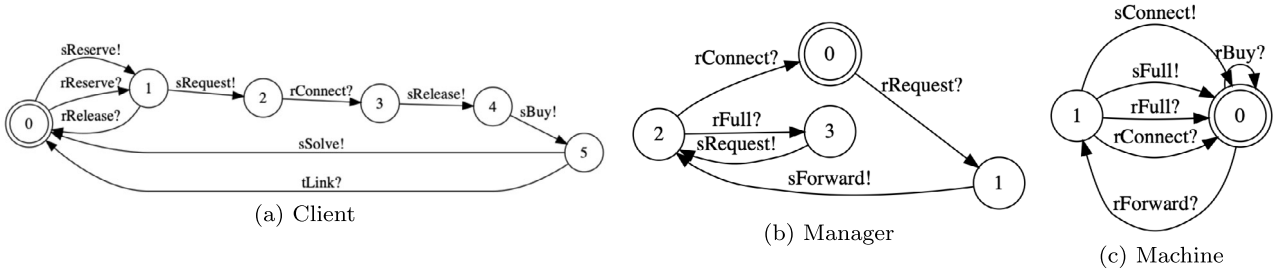
Fig. 5 Machine behaviour

where asgn indicates if the VM is assigned, gLink is a group link, pLink is a private link, and gLink is as before; the initial condition θ_{vm} is $\neg\mathsf{asgn} \wedge \mathsf{cLink} = \mathsf{empty}$ (note gLink and pLink will be machine-specific), where initially VMs are not listening on the common link cLink. Depending on the group that the machine belong to, gLink will be assigned to either high-performance machine group g_1 or the standard one g_2. Moreover, each machine has a unique private link pLink. A VM's send guard is always satisfied (i.e., g^s_{mv} is true), while its receive guard (g^r_{vm}) specifies that it always receives on the broadcast channel and also any channel held in the variables pLink, gLink, and cLink, i.e., g^r_{vm} is

$$\mathsf{ch} = \star \ \vee \ \mathsf{ch} = \mathsf{gLink} \vee \mathsf{ch} = \mathsf{pLink} \vee \ \mathsf{ch} = \mathsf{cLink}$$

The behaviour of the VM agent is reported in Fig. 5. Intuitively, a VM either receives the forwarded request on the group channel gLink (line 2) and thus activates the common link and also a non-deterministic choice between connect and full messages (lines 4–11) or receives a buy message from a client on the private link pLink (line 13). In the latter case, the VM agent agrees to sell the service and stays idle. In the former case, a VM sends connect, with its private link pLink carried on the data variable LNK, on cLink if it is not assigned (line 4), and it sends full on gLink otherwise (line 6). Note that a full message can only go through if all VMs in group gLink are assigned. Note that reception on gLink is always enabled by the receive guard g^r_{vm}. Moreover, the receive transition at line 6 specifies that a machine enables a send on a full message only when it is assigned. For example, if $\mathsf{gLink} = \mathsf{g}_1$, then only when all machines in group g_1 are assigned, a full message can be enabled.

Furthermore, a connect message will also be received by other VMs in the group cLink (line 8). As a result, all other available VMs (i.e., ¬asgn) in the same group do not reply to the request. Thus, one VM is non-deterministically selected to provide a service and a point-to-point-like interaction is achieved. Note that this easy encoding is possible because



(a) Client (b) Manager (c) Machine

Fig. 6 Symbolic structure automata

agents change communication interfaces dynamically by enabling and disabling channels.

Now, we can easily create an R-CHECK system as follows.

$$\begin{aligned}&\mathsf{system} = \mathsf{Client(client1, TRUE)} \parallel \mathsf{Client(client2, TRUE)}\\ &\quad \parallel \mathsf{Client(client3, TRUE)} \parallel \mathsf{Manager(manager, TRUE)}\\ &\quad \parallel \mathsf{Machine(machine1, gLink = g1 \wedge pLink = vmm1)}\\ &\quad \parallel \mathsf{Machine(machine2, gLink = g1 \wedge pLink = vmm2)}\\ &\quad \parallel \mathsf{Machine(machine3, gLink = g2 \wedge pLink = vmm3)}\end{aligned} \tag{1}$$

This system is the parallel composition (according to Definition 2) of three copies of a client $\{\mathsf{client}_1, \ldots, \mathsf{client}_3\}$, a copy of a manager $\{\mathsf{manager}\}$, and finally three copies of a machine $\{\mathsf{machine}_1, \ldots, \mathsf{machine}_3\}$, each belonging to a specific group and a private link. For instance, $\mathsf{machine}_1$ belongs to group g_1 (the high-performance machines) and has a private link named vmm1. The symbolic automata corresponding to the different agents are reported in Fig. 6. There, the interaction structure and the control flow of the different agents are exposed to facilitate a fine-grained reasoning about interactions.

5 Model checking of R-CHECK systems

We present the required background on LTOL, an extension of LTL with the ability to refer to and therefore reason about agent interactions. We also show how to use LTOL to reason about R-CHECK models. In the following sections, we show how to efficiently integrate LTOL model checking into the R-CHECK toolkit.

5.1 The LTOL specification logic

LTOL is an extension of the Linear Time Temporal (LTL) logic with the ability to refer to and therefore reason about agent interactions. LTOL majorly differs from LTL with respect to the next operator (i.e., X). Indeed, LTOL replaces the next operator of LTL with observation descriptors that characterise the contents of the message and the sender predicate. Namely, we distinguish two descriptors, *possible* $\langle O \rangle$ and *necessary* $[O]$, to refer to messages and the intended set of receivers. The syntax of formulas φ and *observation descriptors* O is as follows:

$$\begin{aligned}O &::= pv \mid \neg pv \mid ch \mid \neg ch \mid k \mid \neg k \mid d \mid \neg d \mid \bullet^{\exists} O \mid \bullet^{\forall} O \mid\\ &\quad O \vee O \mid O \wedge O\\ \varphi &::= v \mid \neg v \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \,\mathcal{U}\, \varphi \mid \varphi \,\mathcal{R}\, \varphi \mid \langle O \rangle \varphi \mid [O] \varphi\end{aligned}$$

where pv is a property identifier, ch is a channel name (identifying the channel the current message is sent on), k is an agent identifier (indicating the sending agent at the current time step), and d is a data variable (whose value is determined by the payload of the current message).

We use the classic abbreviations $\rightarrow$ and $\leftrightarrow$ and the usual definitions for true and false. We also introduce the temporal abbreviations $\mathsf{F}\varphi \equiv \mathsf{true}\,\mathcal{U}\,\varphi$ (*eventually*), $\mathsf{G}\varphi \equiv \neg \mathsf{F} \neg \varphi$ (*globally*), and $\varphi \,\mathcal{W}\, \psi \equiv \psi \,\mathcal{R}\, (\psi \vee \varphi)$ (*weak until*). Furthermore, the semantics assumes that all variables mentioned in the specification are boolean. Note that every finite domain can be encoded by multiple boolean variables. R-CHECK, however, supports constraints over non-boolean variables, including variables with infinite domain (e.g., integers) as part of the LTOL syntax.

The syntax of LTOL is presented in *positive normal form* to avoid unnecessary blowup during model checking. That is, we push the negation down to atomic propositions.

Observation descriptors are built from referring to the different parts of the message and their boolean combinations. Thus, they refer to the channel in CH, the data variables in D, the sender k, and the sender predicate over predicate variables in PV. These predicates are interpreted as sets of possible assignments to property identifiers, and therefore we include existential $\bullet^{\exists} O$ and universal $\bullet^{\forall} O$ quantifiers over these assignments.

The semantics of a descriptor O is defined for a ReCiPe message $m = (ch, \mathbf{d}, k, \pi)$. Recall π identifies the set of receivers the sender wishes the message to reach by identifying the assignments to property identifiers PV that a receiver is allowed to have. Given an assignment c and a variable $pv \in \mathrm{PV}$ we write $c \models pv$ if c assigns pv to true and $c \not\models pv$

otherwise. The semantics is defined as follows:

$m \vDash ch'$ **iff** $ch = ch'$	$m \vDash \neg ch'$ **iff** $ch \neq ch'$
$m \vDash d'$ **iff** $\mathbf{d}(d')$	$m \vDash \neg d'$ **iff** $\neg\mathbf{d}(d')$
$m \vDash k'$ **iff** $k = k'$	$m \vDash \neg k'$ **iff** $k \neq k'$

$m \vDash pv$ **iff** for every assignment $c \vDash \pi$ we have $c \vDash pv$
$m \vDash \neg pv$ **iff** there is an assignment $c \vDash \pi$ such that $c \nvDash pv$
$m \vDash \bullet^{\exists} O$ **iff** there is an assignment $c \vDash \pi$ such that $(ch, d, k, \{c\}) \vDash O$
$m \vDash \bullet^{\forall} O$ **iff** for every assignment $c \vDash \pi$ we have $(ch, d, k, \{c\}) \vDash O$
$m \vDash O_1 \vee O_2$ **iff** either $m \vDash O_1$ or $m \vDash O_2$
$m \vDash O_1 \wedge O_2$ **iff** $m \vDash O_1$ and $m \vDash O_2$

We only comment on the semantics of the descriptors $\bullet^{\exists} O$ and $\bullet^{\forall} O$ as the rest are standard propositional formulas. The descriptor $\bullet^{\exists} O$ requires that at least one assignment c to the property identifiers in the sender predicate π satisfies O. Dually $\bullet^{\forall} O$ requires that all assignments in π satisfy O. Using the former, we express properties where we require that the sender predicate has a possibility to satisfy O, while using the latter, we express properties where the sender predicate can only satisfy O. For instance, both observations $(ch, \mathbf{d}, k, pv_1 \vee \neg pv_2)$ and $(ch, \mathbf{d}, k, pv_1)$ satisfy $\bullet^{\exists} pv_1$, while only the latter satisfies $\bullet^{\forall} pv_1$. Furthermore, the observation descriptor $\bullet^{\forall}\,\mathsf{false} \wedge ch = \star$ says that a message is sent on the broadcast channel with a false predicate. That is, the message cannot be received by other agents. In our example in Sect. 4, the descriptor $\bullet^{\exists}(\mathsf{pv} = \mathsf{client}) \wedge \bullet^{\forall}(\mathsf{pv} = \mathsf{client})$ says that the message is intended exactly for agents with a client role.

We interpret LTOL formulas over system computations.

Definition 5 (System computation)
A system computation ρ is a function from natural numbers N to $2^{\mathcal{V}} \times M$ where $\mathcal{V}$ is the set of state variable propositions and $M = \mathrm{CH} \times 2^{\mathrm{D}} \times K \times 2^{2^{\mathrm{PV}}}$ is the set of possible observations. That is, ρ includes values for the variables in $2^{\mathcal{V}}$ and a message in M at each time instant.

We denote by s_i the system state (i.e., an assignment to system variables) at the i-th time point of the system computation. Moreover, we denote the suffix of ρ starting with the i-th state by $\rho_{\geq i}$ and we use m_i to denote the message $(ch, \mathbf{d}, k, \pi)$ in ρ at time point i.

The semantics of an LTOL formula φ is defined for a computation ρ at a time point i as follows:

$\rho_{\geq i} \vDash v$ **iff** $s_i \vDash v$ and $\rho_{\geq i} \vDash \neg v$ **iff** $s_i \nvDash v$;
$\rho_{\geq i} \vDash \varphi_2 \vee \varphi_2$ **iff** $\rho_{\geq i} \vDash \varphi_1$ or $\rho_{\geq i} \vDash \varphi_2$;
$\rho_{\geq i} \vDash \varphi_2 \wedge \varphi_2$ **iff** $\rho_{\geq i} \vDash \varphi_1$ and $\rho_{\geq i} \vDash \varphi_2$;
$\rho_{\geq i} \vDash \varphi_1 \,\mathcal{U}\, \varphi_2$ **iff** there exists $j \geq i$ s.t. $\rho_{\geq j} \vDash \varphi_2$ and for every $i \leq k < j$, $\rho_{\geq k} \vDash \varphi_1$;
$\rho_{\geq i} \vDash \varphi_1 \,\mathcal{R}\, \varphi_2$ **iff** for every $j \geq i$ either $\rho_{\geq j} \vDash \varphi_2$ or there exists $i \leq k < j, \rho_{\geq k} \vDash \varphi_1$;
$\rho_{\geq i} \vDash \langle O \rangle \varphi$ **iff** $m_i \vDash O$ and $\rho_{\geq i+1} \vDash \varphi$;
$\rho_{\geq i} \vDash [O]\varphi$ **iff** $m_i \vDash O$ implies $\rho_{\geq i+1} \vDash \varphi$.

Intuitively, the temporal formula $\langle O \rangle \varphi$ is satisfied on the computation ρ at point i if the message m_i satisfies O *and* φ is satisfied on the suffix computation $\rho_{\geq i+1}$. On the other hand, the formula $[O]\varphi$ is satisfied on the computation ρ at point i if m_i satisfying O *implies* that φ is satisfied on the suffix computation $\rho_{\geq i+1}$. Other formulas are interpreted exactly as in LTL.

With observation descriptors we can refer to the intention of agents in the interaction. Consider the following formula:

$$\bigvee_{\mathsf{k} \in \mathsf{Client}} \mathsf{F}\langle \mathsf{sender} = \mathsf{k} \wedge \bullet^{\exists}(\mathsf{pv} = \mathsf{mgr}) \wedge \mathsf{MSG} = \mathsf{request} \rangle \mathsf{true}$$

The formula states that eventually a client k will communicate with the resource manager mgr using a request message. Note that we not only specify the message contents in LTOL, but we can also predicate on the targeted receivers. Expressing this formula in LTL, which was the only specification language supported in the conference version [6] of this article, is only possible by including (manually) additional instrumentation in the system. Thus, we have integrated LTOL model checking into R-CHECK. Here, we would like to use existing implementations of LTL model checking rather than the bespoke LTOL model-checking algorithm in [1]. The algorithm here replaces the large alphabet required in [1] by extending the model with observation variables (to be explained below) and creating a modified LTL formula relating to them. Thus, we reduce LTOL model checking to LTL model checking over a model that is extended with additional variables. The details are given in Sect. 6.2.

Practically, the implementation of the latter algorithm is done through an encoding into the NUXMV model checker [15]. First, we transform an R-CHECK model into a symbolic ReCiPe model, we encode it as an NUXMV module, and later we rely on the symbolic model-checking algorithms of NUXMV to reason about R-CHECK.

5.2 LTOL showcasing

We will use Eq. (1) (Sect. 4) and the corresponding structure automata in Fig. 6 as the system under consideration.

We will show how to verify LTOL properties of agents at both individual and interaction protocol levels by predicating on message exchange rather than on atomic propositions. Unlike the conference version [6], we will use formulas that natively characterise interactions and the different coordination mechanisms. For instance, we can reason about a client and its connection to the system as follows.

$$\mathsf{G}([\mathsf{sender} = \mathsf{client1} \wedge \mathsf{MSG} = \mathsf{reserve}] \; \mathsf{F}\, \langle \mathsf{sender} = \mathsf{client1} \wedge \mathsf{MSG} = \mathsf{request} \rangle \mathsf{true}) \quad (1)$$

$$\mathsf{G}([\mathsf{sender} = \mathsf{client1} \wedge \mathsf{MSG} = \mathsf{reserve}] \; \mathsf{F}\, \langle \mathsf{sender} = \mathsf{client1} \wedge \mathsf{MSG} = \mathsf{release} \rangle \mathsf{true}) \quad (2)$$

$$\mathsf{G}([\mathsf{sender} = \mathsf{client1} \wedge \mathsf{MSG} = \mathsf{request}] \; \mathsf{F}\, \langle \mathsf{MSG} = \mathsf{connect} \rangle \mathsf{client1{-}mLink} \neq \mathsf{empty}) \quad (3)$$

The liveness condition (1) specifies that after a client reserves the common link they send a request to the manager c. The liveness condition (2) specifies that the client does not hold a live lock on the common link c, namely, the client releases the common link eventually. The liveness condition (3) specifies that the *system* is responsive, i.e., after the client's request, other agents collaborate to eventually supply a connection.

We can also reason about synchronisation and reconfiguration in relation to local state as follows.

$$\mathsf{G}([\mathsf{sender} = \mathsf{manager} \wedge \mathsf{MSG} = \mathsf{request}] \bigvee_{\mathsf{k} \in \mathsf{Machine}} \mathsf{k{-}cLink{=}c}) \quad (4)$$

$$\bigwedge_{\mathsf{k} \in \mathsf{Machine}} \mathsf{G}\,(!\mathsf{k{-}asgn} \rightarrow \mathsf{F}\, \langle \mathsf{sender} = \mathsf{k} \wedge \mathsf{MSG} = \mathsf{connect} \rangle \mathsf{true}) \quad (5)$$

In (4), the manager has to forward the request before a machine can get connected to the common link. That is, a machine reconfigures its interaction interface and starts listening to link c. Moreover, in (5) every machine that is not assigned must eventually supply a connection.

We can also specify channel mobility and joint missions from a declarative and centralised point of view, as follows.

$$\bigvee_{\mathsf{k} \in \mathsf{Client}} \mathsf{F} \left(\langle \mathsf{sender} = \mathsf{k} \wedge \mathsf{MSG} = \mathsf{complete} \rangle \mathsf{true} \wedge \bigwedge_{\mathsf{j} \neq \mathsf{k} \in \mathsf{Client}} [\mathsf{sender} \neq \mathsf{j} \wedge \mathsf{MSG} = \mathsf{complete}]\mathsf{j{-}rSolve} \right) \wedge \bigwedge_{\mathsf{k} \in \mathsf{Client}} \mathsf{F}\; \mathsf{k{-}mLink} \neq \mathsf{empty}$$

That is, eventually one client will initiate the mission's termination by synchronising with the others to solve the joint problem. Notice that the quantified subformula over clients (in the second line) that are not senders ensures that they must participate and supply a receive transition (i.e., j−rSolve; see Fig. 3, line 23). Moreover, each client eventually receives a mobile link (i.e., k-mLink ≠ empty).

This is interesting because we can zoom in and specify senders and receivers natively. Indeed, this level of reasoning is impossible to achieve while considering only the states of the system. We must be able to refer to contents of messages as allowed by LTOL. The full tool support and all examples in this paper are available in the git repository.

In the following, we will show how to integrate LTOL into R-CHECK.

6 LTL and LTOL model checking and simulation

We will show how to model check both LTL and LTOL [1, 5] through NUXMV. With NUXMV, we can support BDD-based symbolic model checking (with finite state-space) or IC3 and BMC (with infinite state-space). Moreover, we present a new native frontend and interpreter for R-CHECK and showcase it.

6.1 Integrating LTL model checking into R-CHECK

We give individual R-CHECK agents a symbolic semantics based on the ReCiPe framework as shown in Sect. 3.1 and Definition 4. Notably, we preserve the labels of commands (i.e., $l : \sigma$) and use them as subpredicate definitions. For instance, given a labelled edge $(s_1, l : \sigma, s_2)$ in the structure automaton G in Definition 3, we translate it into the following predicate in ReCiPe as explained in Definition 4:

$$l := \mathsf{predOf}(\sigma) \wedge (\mathsf{st} = s_1) \wedge (\mathsf{st}' = s_2) \wedge \textsc{keep}(V_T \setminus \mathsf{varsOf}(\sigma))$$

The only difference here is that the label l is now a predicate definition and its truth value defines if the transition $(s_1, l : \sigma, s_2)$ is feasible. Since every command is translated to either message-send or message-receive, we can use these labels now to syntactically refer to code reachability. That is, we can directly specify if a line of code is reachable.

Moreover, we rename all local variables of agents to consider the identity of the agent as follows: for example, given the cLink variable of a client, we generate the variable client−cLink. This is important when different agents use the same identifier for local variables. We also treat all data variables D and channel names CH as constants and we construct a ReCiPe system $S = \langle \mathcal{V}, \rho, \theta \rangle$ as defined in Definition 2 while considering subpredicate definitions and agent variables after renaming. Technically, a ReCiPe system S has a one-to-one correspondence to a NUXMV module M. That is, both S and M agree on local variables $\mathcal{V}$ and the initial condition θ, but are slightly different with respect to

transition relations. For efficiency, the composition of agents to a system discards the information about messages. Thus, we do *not* add variables that capture these details and rely on correct conjunction of transition disjuncts to capture the *existence* of an appropriate message. Finally, the transition relation ρ of S as defined in Definition 2 is translated to an equivalent transition relation $\hat{\rho}$ of M as follows:

$$\hat{\rho} = \rho \vee (\neg\rho \wedge \textsc{keep}(\mathcal{V}))$$

That is, NuXmv translates deadlock states in S into stuttering (sink) states in M where system variables do not change.

Note that the abovementioned translation to NuXmv unlocks the native model simulator and LTL model checker of NuXmv. As messages are not encoded directly in the model, only basic reasoning about message exchange using labelled commands is possible. As shown in the early version [6] of this article, one can use such labels to reason about interactions as follows:

$$\mathsf{G}\,(\mathsf{client1{-}sRequest} \rightarrow \mathsf{F}\,\mathsf{client1{-}rConnect})$$

This formula can be used in our example to specify that after the client's request eventually a machine supplies a connection. Although this formula is correct in our example, it is not strong enough to indicate that a request message is actually happening. It can only indicate that a request message is enabled. This is because the label sRequest only indicates if a line of code is reachable and feasible, but does not tell if it is executed. One needs a more expressive language to express the actual exchange of messages. For instance, this is straightforward in the following LTOL formula:

$$\varphi \triangleq \mathsf{G}([\mathsf{sender} = \mathsf{client1} \wedge \mathsf{MSG} = \mathsf{request}] \; \mathsf{F}\,\langle \mathsf{MSG} = \mathsf{connect}\rangle \mathsf{client1{-}mLink} \neq \mathsf{empty}) \quad (1)$$

Clearly, Eq. (1) natively expresses that it is always the case that if a client sends a request message, it will eventually receive a connection where its mobile link mLink is assigned. However, to allow such high-level verification of message exchange, we need to integrate LTOL model checking into R-CHECK.

6.2 Integrating LTOL model checking into R-CHECK

We provide a new algorithm for model checking LTOL. The original algorithm [1] constructed directly a corresponding automaton from an LTOL specification. Furthermore, during model checking additional satisfiability checks of individual steps were required. Here, to reuse existing state-of-the-art tools, we instead augment our system model to be able to project LTOL model checking onto LTL model checking. Note that this is not equivalent to an encoding of LTOL into LTL. Such encoding does not actually exist, and this is why our algorithm requires augmenting the model with new variables.

Our algorithm exploits the original insights of the algorithm for LTOL model checking [1], namely, to concentrate on checking messages against top-level observations, appearing in the LTOL formula.

A core difficulty in the present algorithm is that the observation descriptors O that appear in LTOL formulas are not (completely) part of R-CHECK models. Therefore, we need a way to embed them efficiently in R-CHECK models and use this embedding to reason about interactions through NuXmv. To do so, we introduce a variable obs_i for each descriptor that appears in the LTOL formula, embed obs_i into the NuXmv encoding of the R-CHECK model, and later use an extended LTL formula to reason about it. We stress that the embedding of obs_i into the NuXmv encoding ensures that the value of obs_i reflects the truth value of the observation after each transition.

Formally, let $\mathsf{obs}(\varphi)$ be the set of observations appearing "top-level" in the operators $\langle\cdot\rangle$ and $[\cdot]$ in φ. More precisely, $\mathsf{obs}(\varphi)$ is closed under the subformula relation of φ, but is not closed under the subformula relation of O. Consider φ in Eq. (1):

$$\mathsf{obs}(\varphi) = \{\mathsf{sender} = \mathsf{client1} \wedge \mathsf{MSG} = \mathsf{request}, \; \mathsf{MSG} = \mathsf{connect}\}$$

We denote by $|\mathsf{obs}(\varphi)|$ the size of the set $\mathsf{obs}(\varphi)$. Thus, for a formula φ over an R-CHECK system Sys, we create a modified system Sys' by introducing $|\mathsf{obs}(\varphi)|$ new variables (one for each descriptor) to the system. These variables can be used to record the changes of truth values to the different observation descriptors. We set the initial condition θ' of the modified system Sys' to the conjunction of the initial condition θ of the original system Sys and a false assignment to all these new descriptor variables, to mean that no messages have been exchanged initially.

Recall that the predicate semantics of an R-CHECK system is of the form $\bigvee_i(\bigwedge_j \mathsf{cmd}_j)$, where each conjunct in the disjunction represents a message-send command in $\mathcal{T}_i^s$ conjuncted with reactions of receivers in $\mathcal{T}_j^r$ with respect to g_j^r, while evaluating the send predicate g_i^s on each receiver state. This means that it is sufficient to set the new descriptor variables after the execution of each conjunct to specify which message is executed. In what follows, we abuse the notations and use $\mathcal{T}_i^s$ to denote the set of send transitions of agent i and τ_i^s to denote a single send transition of agent i.

In other words, for each conjunct $(\bigwedge_j \mathsf{cmd}_k)$ in the system semantics, we conjunct it with the truth value of each descriptor in the next state. Namely, the latter conjunct is transformed into

$$(\bigwedge_j \mathsf{cmd}_j) \wedge \bigwedge_k (\mathsf{obs}'_{\mathsf{k}} = \mathsf{embed}(\mathsf{O}_{\mathsf{k}}, \tau^{\mathsf{s}}_{\mathsf{j}})) \quad (2)$$

That is, every time a message is emitted, we assign each obs_k with its truth value in the next state (hence we use the prime copy obs'_k) through an embedding function. The latter takes the descriptor O_k for sender k and the send transition τ_j^s as parameters. The definition of the embed function is shown below.

The rationale is that each transition at the system level is a send transition, which is originated by a single sender. Thus, every time a message is emitted, we set the truth values of all observation descriptors. We embed the observations for each send transition $\tau_k^s \in \mathcal{T}_k^s$ of agent k. The embedding function is defined as follows.

Definition 6 (Observation embedding)
We define an embedding function $\mathsf{embed}(O_k, \tau_j^s)$ that takes an observation O_k and a send transition τ_j^s of agent j and rewrites the observation to a formula with respect to τ_j^s and the send predicate g_j^s of agent j. We use c to denote an assignment to property identifiers in PV and $f[c]$ to denote a grounding of formula f on c:

$$
\begin{aligned}
&\mathsf{embed}(k, \tau_j^s) && \triangleq k = j\\
&\mathsf{embed}(ch, \tau_j^s) && \triangleq ch = ch(\tau_j^s)\\
&\mathsf{embed}(d, \tau_j^s) && \triangleq d = \mathbf{d}_{\tau_j^s}(d)\\
&\mathsf{embed}(pv, \tau_j^s) && \triangleq pv\\
&\mathsf{embed}(\neg O, \tau_j^s) && \triangleq \neg\mathsf{embed}(O, \tau_j^s)\\
&\mathsf{embed}(\bullet^{\exists} O, \tau_j^s) && \triangleq \bigvee_{c \in \mathrm{PV}} (g_s[c] \wedge \mathsf{embed}(O, \tau_j^s)[c])\\
&\mathsf{embed}(\bullet^{\forall} O, \tau_j^s) && \triangleq \bigwedge_{c \in \mathrm{PV}} (g_s[c] \rightarrow \mathsf{embed}(O, \tau_j^s)[c])\\
&\mathsf{embed}(O_1 \wedge O_2, \tau_j^s) && \triangleq \mathsf{embed}(O_1, \tau_j^s) \wedge \mathsf{embed}(O_2, \tau_j^s)\\
&\mathsf{embed}(O_1 \vee O_2, \tau_j^s) && \triangleq \mathsf{embed}(O_1, \tau_j^s) \vee \mathsf{embed}(O_2, \tau_j^s)
\end{aligned}
$$

Namely, we rewrite the observation in relation to the current executed send command and the sender predicate. We require that LTOL formulas are written in a normal form where PVs do not appear outside quantifiers and there is no nesting of quantifiers. This is important to handle quantified formulas correctly.

Now, everything is in place to translate LTOL descriptor formulas into LTL as follows:

$$
\begin{aligned}
[\![\langle O \rangle \varphi]\!] &\triangleq \mathsf{X}(\mathsf{obs}_O \wedge [\![\varphi]\!])\\
[\![[O] \varphi]\!] &\triangleq \mathsf{X}(\mathsf{obs}_O \rightarrow [\![\varphi]\!])
\end{aligned}
\tag{3}
$$

Intuitively, the translation faithfully follows the trace semantics of descriptor formulas as explained in Sect. 5.1. Note that the resulting LTL formula is linear in size with respect to the LTOL one. Here, we consider the size of an LTOL formula with respect to top-level observations as in [1], and thus we consider an observation alphabet in $2^{|\mathsf{obs}(\varphi)|}$. Since we eventually employ LTL model checking on the modified model and formula, it is clear that the model-checking problem is still in PSPACE. However, the size of the model is dependent on the extra variables added to account for top-level observations $|\mathsf{obs}(\varphi)|$. More precisely, we have the following.

Theorem 1 (Model checking)
The model-checking problem of an LTL *formula φ' translated from an* LTOL *formula φ is* PSPACE*-complete with respect to the size of the original* LTOL *formula $|\varphi|$ and the size of the modified system $|Sys| \times |\mathsf{obs}(\varphi)|$, where $|Sys|$ is the (symbolic) size of the original system.*

Note that the stated bound in terms of $|Sys| \times |\mathsf{obs}(\varphi)|$ instead of LOGSPACE is because R-CHECK systems are symbolic. Indeed, LOGSPACE complexity is achieved for enumerative representation, which is anyway exponentially larger.

Clearly, the size of the resulting LTL formula (which is linear compared to LTOL) does not play a role in the asymptotic complexity. The only major difference is due to the enriched system model. Since the number of top-level observations in the formula $|\mathsf{obs}(\varphi)|$ impacts the size of the model, one could mitigate the blowup by model checking individual formulas, and thus building smaller enriched system models for each LTOL formula.

Theorem 2 (Correctness)
Given a ReCiPe system Sys and an LTOL *formula φ, we have*

$$Sys \vDash \varphi \quad \textit{if and only if} \quad Sys' \vDash \varphi'$$

where Sys' is an enriched system model according to Eq. (2) and Definition 6 and φ' is an LTL *formula translated from φ according to Eq. (3).*

Proof

We have two directions. We prove the direction ($\Rightarrow$), and the direction ($\Leftarrow$) follows in a similar way. Moreover, we restrict our attention to base and descriptor formulas. Note that the translation of other formulas is the identity function.

($\Rightarrow$) Assume $Sys \vDash \varphi$ and prove $Sys' \vDash \varphi'$:

For a ReCiPe system Sys, a computation is of the form $\sigma : (s_0, m_0)(s_1, m_1) \ldots$, where s_0 is an initial state, $s_i \in 2^{\mathcal{V}}$, and $m_i \in M$ (see Definition 5). An R-CHECK system computation, on the other hand, is of the form $\sigma' : s_0, s_1, \ldots$, where information about messages (m_i) is dropped after composition to produce a NUXMV module. Our algorithm aims at enriching the states of a computation when needed to allow reasoning about message exchange.
We know that $Sys \vDash \varphi$ iff for every computation σ in Sys we have $\sigma \vDash \varphi$. Now consider an arbitrary ReCiPe computation $\sigma : (s_0, m_0)(s_1, m_1) \ldots$ of Sys, and consider the following cases for φ:

– Consider a state formula (v): A state formula does not contain observation descriptors, and thus $|\mathsf{obs}(v)| = 0$.

In other words, both the system Sys and the formula φ are not changed by our algorithm, and thus this case holds easily (state information is sufficient to prove φ).

– Consider a descriptor formula ($\langle O\rangle\varphi$):
By the semantics of LTOL, we have

$$\sigma_{\geq i} \vDash \langle O\rangle\varphi \ \textbf{iff}\ m_i \vDash O \ \text{ and } \ \sigma_{\geq i+1} \vDash \varphi$$

By our algorithm in Sect. 6.2, $|\mathsf{obs}(\langle O\rangle\varphi)| = 1$ (i.e., there is only one top-level observation O), and thus we introduce a new state variable obs_O that stores the satisfiability of O with respect to a previously taken transition (see Eq. (2)).

The modified R-CHECK system Sys' has computations of the form $\sigma' : (s_0, o_0)(s_1, o_1)\ldots$, where $o_i \in 2^{\mathsf{obs}_O}$ for $i \geq 0$ and $o_0 = \emptyset$ (no messages have been exchanged earlier). Note that o_i in a computation (except for o_0) evaluates to true iff the predicate $\mathsf{embed}(\mathsf{O}, \tau^{\mathsf{s}}_{\mathsf{k}})$ is satisfiable for the previously executed send command τ^s_k of some sender k (recall that the embed is assigned as the next assignment to obs_O (i.e., obs'_O)).

By inspection, it is not hard to see that the semantics of $m_i \vDash O$ is actually equivalent to the satisfiability of the predicate returned by $\mathsf{embed}(\mathsf{O}, \tau^{\mathsf{s}}_{\mathsf{k}})$ (or the value of obs'_O). In other words, $m_i \vDash O$ implies o_{i+1} is true.

Now, $\varphi' \triangleq \mathsf{X}(\mathsf{obs}_O \wedge [\![\varphi]\!])$. Thus, for a modified R-CHECK computation $\sigma' : (s_0, o_0)(s_1, o_1)\ldots$, we have $\sigma'_{\geq i} \vDash \mathsf{X}(\mathsf{obs}_O \wedge \varphi)$ iff $\sigma'_{\geq i+1} \vDash \mathsf{obs}_O$ and $\sigma'_{\geq i+1} \vDash [\![\varphi]\!]$.

$\sigma'_{\geq i+1} \vDash \mathsf{obs}_O$ iff obs_O is satisfiable and this is implied by $m_i \vDash O$.

$\sigma'_{\geq i+1} \vDash [\![\varphi]\!]$ follows by $\sigma_{\geq i+1} \vDash \varphi$ and the induction hypothesis.

- Consider a descriptor formula ($[O]\varphi$): This case is similar to the previous one. □

Simulation R-CHECK provides an interactive interpreter (or simulator) that allows the user to simulate the system either randomly or based on the user choice. The simulator can also backtrack from a specific state of the system. The latter feature is used to simulate the counterexamples from the model-checking algorithm. Unlike the NUXMV simulator used in the conference version [6] of this article, the current interpreter is developed based on the interaction semantics of ReCiPe and thus allows to simulate the interactions and provides better understanding of the scenario under consideration.

Note that in R-CHECK, we limit LTOL specifications to only refer to property identifiers of finite domains (e.g., boolean variables, channels, enums, and bounded integers). This is important because the definition of embed requires, for certain cases, existential (e.g., for $\bullet^{\exists}$) or universal (for $\bullet^{\forall}$) quantification over the possible assignments to PV, and thus we may need to enumerate all such assignments.

6.3 The R-CHECK frontend and interpreter

R-CHECK is implemented in a prototype tool, which can be invoked from either a command line or a user-friendly web interface (with graphical illustrations).

The web interface (e.g., Fig. 7) provides a rich text area and support for model checking or simulation.

The text area permits writing high-level syntax corresponding to the language of R-CHECK, providing syntax colouring and highlighting to improve readability. The user can build the model by compiling either to an SMT model (with infinite state-space) or to a plain BDD (with finite state-space) by hitting the Build model button. When the model is ready, the user will also be provided with a representation of the agents in the system as symbolic automata, shown in the lower part of the interface (see Fig. 7).

For model checking purposes, the user can use different options, depending on the model type. Currently, we support BDD-based symbolic model checking (which is Model Checking (MC), which requires a BDD model), IC3, and BMC (which require an SMT model). When either IC3 or BMC is selected, a text field appears where a verification bound can be specified; this is optional for IC3 and mandatory for BMC.

The user writes all specifications at the bottom of the text area, and each specification should be prefixed by the keyword SPEC. Once the model checking procedure is over, the user gets a report with the verification outcome for each specification.

The interpreter tab allows the user to interactively explore the system's executions. We set up the interpreter by hitting the Start button. When this happens, the initial state is computed,[4] a dropdown menu gets populated with all available transitions, and the Start button itself is replaced by a Next button. Then, the user makes the system evolve by choosing a transition from the dropdown menu and hitting Next. The Back button allows instead to undo the latest transition and go back to the previous step. The interpreter can be restarted completely by hitting Reset. Additionally, whenever a verification task (in the Model Checking tab) finds a counterexample, the user will be given the option to load it into the interpreter.

[4] To better handle systems with non-determinism in the initial state, we plan to add an input field where the user can specify initial constraints.

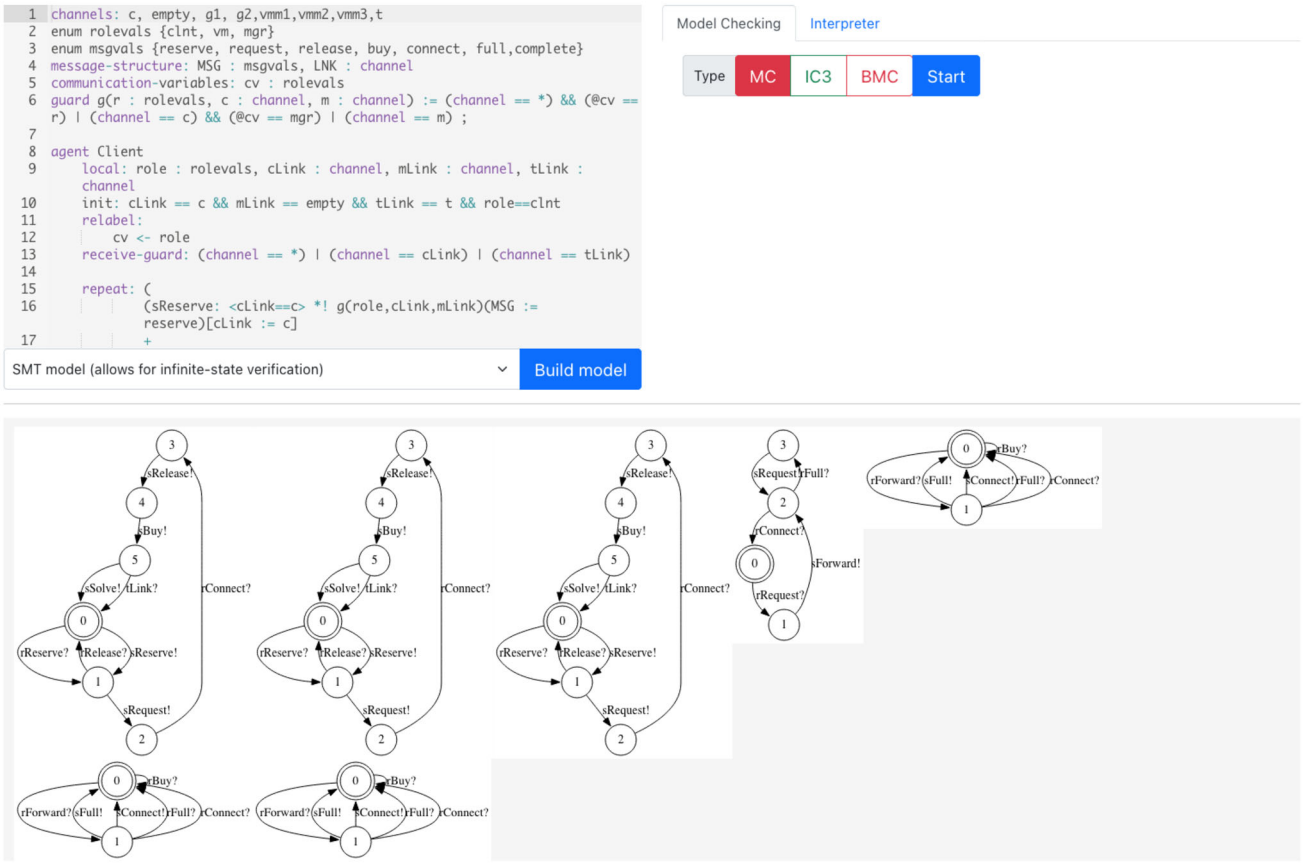


Fig. 7 The R-CHECK Web-based interface, after building a model

Let us now consider the R-CHECK example of Sect. 4 and show how to use the Web frontend to model check the following specifications against it:

$$\bigvee_{k \in \text{Client}} F\langle \text{sender} = k \wedge \text{MSG} = \text{complete}\rangle \text{true} \wedge \bigwedge_{k \in \text{Client}} F\ k\text{–mLink} \neq \text{empty} \quad (1)$$

$$G(\langle \text{sender} = \text{manager} \wedge \text{MSG} = \text{request}\rangle \text{true} \rightarrow \bigwedge_{k \in \text{Machine}} [\text{sender} = \text{manager}]k\text{–cLink}=c) \quad (2)$$

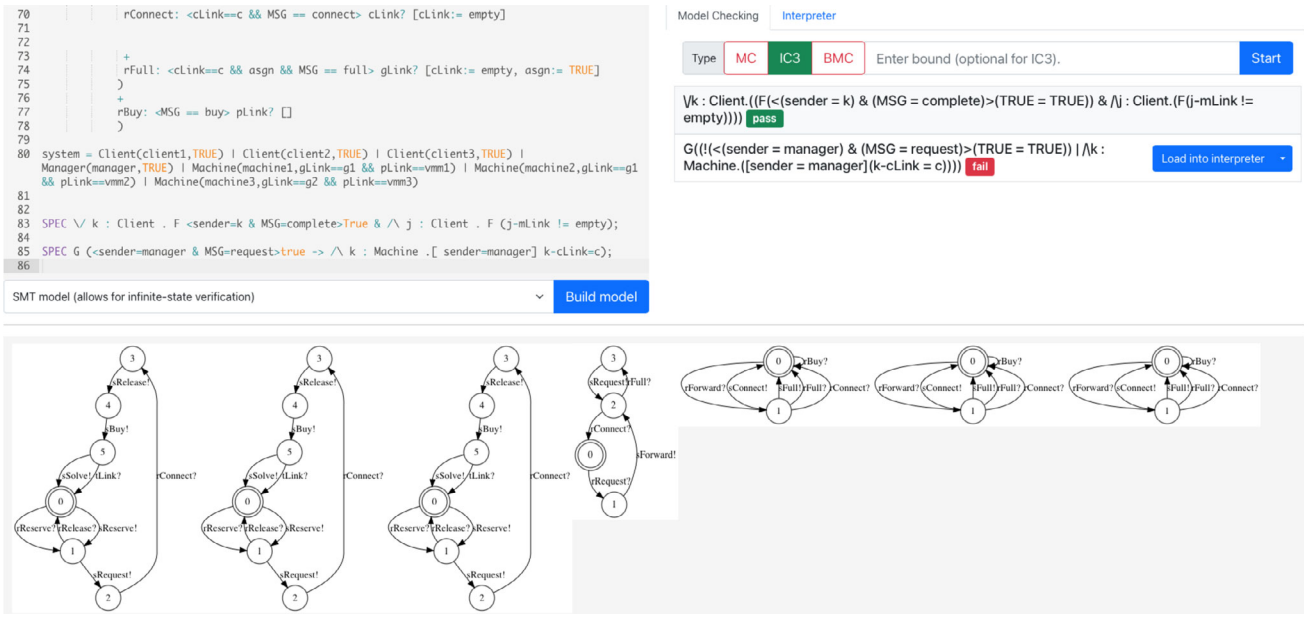
We expect property (1) to hold and (2) to be violated. Informally, (1) states that eventually one client will initiate the mission's termination. Moreover, each client eventually receives a mobile link. On the other hand, (2) states that once the manager has forwarded the request, all machines will get connected to the common link.

Figure 8a depicts the outcome of the IC3 model checker. As we can see, the tool allows us to load the counterexample for the second property into the interpreter. When we do that (see Fig. 8b), we can easily see that after the manager sends a request message from state 2, only machine1 and machine2 get connected to link c. This is not the case for machine3, thus violating property (2).

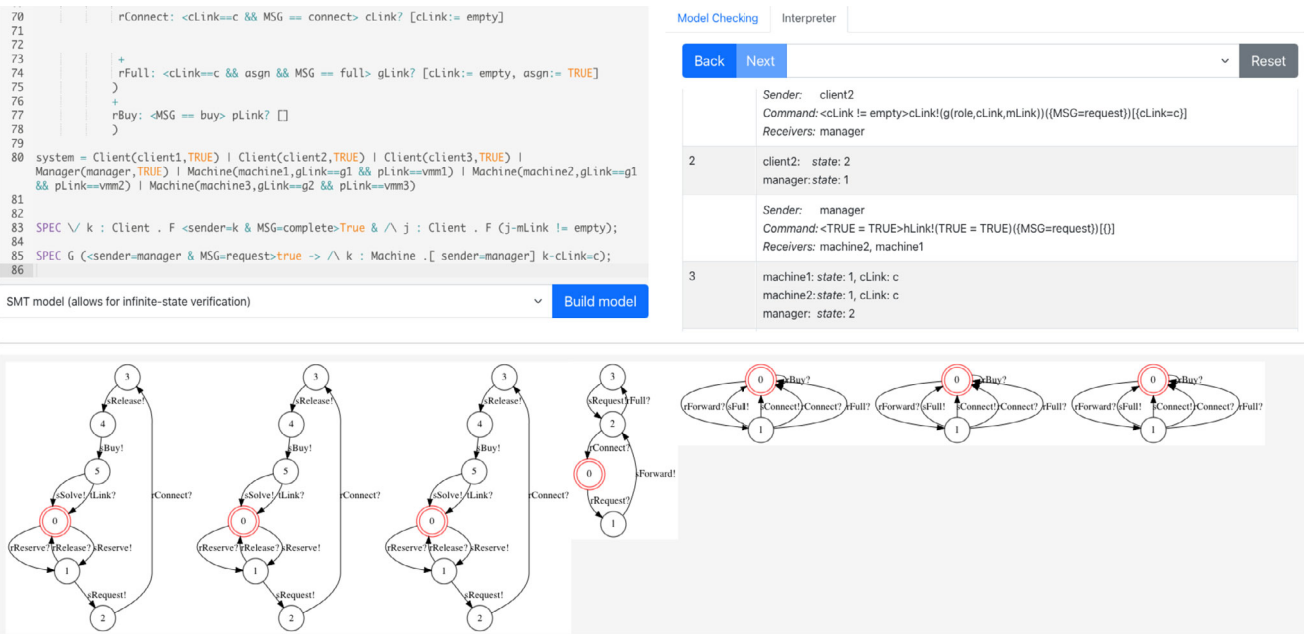
7 Concluding remarks

We introduced the R-CHECK model-checking toolkit for verifying and simulating reconfigurable MASs. We formally presented the syntax and semantics of R-CHECK in relation to the ReCiPe framework [1, 5], and we used it to model and reason about a non-trivial case study from the realm of reconfigurable and self-organising MASs. Our semantics approach consisted of two types of semantics: structural semantics in terms of automata to recover information about interaction features and execution semantics based on ReCiPe. The interaction information recovered in the structural semantics is recorded succinctly in the execution one and thus permits reasoning about interaction protocols and message exchange. R-CHECK is supported with a command line tool, a web editor with syntax highlighting, and visualisation.

We integrated LTOL [1, 5] model checking intoR-CHECK, thus allowing native reasoning about selective interaction strategies. The integration consists of providing a dynamic embedding of LTOL descriptor formulas into the model under consideration. We built R-CHECK based on a compilation to NUXMV to enable both LTOL and LTL verification through symbolic, bounded, and IC3 model checking.



(a) Model checker tab with verification outcome.



(b) Interpreter with a counter example to a violated specification.

Fig. 8 Using R-CHECK to analyze the system of Sect. 4

We showed that this specialised integration provides a powerful tool that permits verifying high-level features such as synchronisations, interaction protocols, joint missions, channel mobility, reconfiguration, and self-organisation.

As mentioned, our work is focused on MASs, which are a special type of CASs. The difference here is that the number of agents is usually small, and thus the issue of scalability is not our main concern. Indeed, if we consider a large number of agents, then qualitative reasoning with LTOL would not be sufficient and probabilistic techniques, like statistical model checking [30], would be more appropriate.

Related works We report on closely related model-checking toolkits.

MCMAS [31] is a successful model checker that is used to reason about MASs and supports a range of temporal and epistemic logic operators. It also supports ISPL, a high-level input language with semantics based on *interpreted*

systems [23]. The key differences with respect to R-CHECK are: (1) MCMAS models are enumerative and are exponentially larger than R-CHECK ones; (2) actions in MCMAS are merely synchronisation labels while command labels in R-CHECK are predicates with truth values changing dynamically at run-time, introducing opportunistic interaction; and (3) lastly and most importantly, R-CHECK can model and reason about dynamic communication structures with message exchange and channel mobility while in MCMAS the structure is fixed.

The MTSA toolkit [22] is used to reason about labelled transition systems (LTSs) and their composition as in the simple multi-way synchronisation of Hoare's CSP calculus [27]. MTSA uses *Fluent Linear Temporal Logic* (FLTL) [25] to reason about actions, where a fluent is a predicate indicating the beginning and the end of an action. As the case of MCMAS, the communication structure is fixed and there is no way to reason about reconfiguration or even message exchange.

A few other languages have been proposed that entirely drop channel-based interaction, letting agents select their interaction partners through attribute-based predicates. Here we only report on those languages with an associated verification platform. In SCEL [19], each agent (or *process*) has an associated *tuple space* and interaction happens by attribute-based insertion, lookup, or deletion of tuples. This makes the interaction mechanism somehow asynchronous, in the sense that (i) the insertion of a tuple cannot be blocked and (ii) there is no guarantee that a tuple insertion modelling a service request will elicit a response within any time bound. While SCEL is arguably a more dynamic language than R-CHECK, featuring dynamic instantiation of names and processes as well as higher order communication (i.e., exchanging processes by storing them in tuples), its verification capabilities are based on statistical model checking [35]. This is due to the fact that SCEL's models have typically infinite state-space, both behaviourally (dynamic creation of processes) and domain-wise (using infinite-domain state variables). Thus, statistical reasoning fits more with SCEL.

AbC [3, 4] instead provides attribute-based multi-way synchronisation as the core interaction primitive. *AbC* specifications may be verified through symbolic BMC [21]. This approach seems limited to verification of safety properties and appears to be better suited for bug finding than for liveness and fairness properties, since completeness of BMC depends on choosing an appropriate verification bound. Compared to SCEL and *AbC*, R-CHECK offers channel-based communication that may be further specialised through predicates over properties. These, in turn, bear a loose resemblance to attributes. Properties appear to be more flexible and to better support encapsulation: the value of a property is the result of an arbitrary expression over the state of agents, whereas attributes either coincide with (*AbC*) or directly map to internal variables (SCEL). At the same time, properties are not essential to the reconfigurable MAS aspects of R-CHECK. For instance, agents may still block each other over multicast channels even without send guards.

Other frameworks that deal with dynamic reconfiguration include DREAM [20] and BIP [12]. In the former, components (agents) live within *motifs* that also dictate rules for components to interact with each other or migrate towards another motif. In the latter, behaviour and interactions are logically distinct layers. The behavioural layer only specifies how a component communicates over a set of ports; the interaction layer, in turn, specifies *connectors* that model links and modes of synchronisation between ports. This modelling style is known as *exogenous*, as opposed to the *endogenous* style where coordination primitives are part of the components' behaviour. Proponents of exogenous modelling argue that it enables abstract formal analysis of the coordination model away from the behavioural layer. ReCiPe applies instead an endogenous approach: the rationale is that we are dealing with highly dynamic and reconfigurable systems. The latter makes compositional verification (in terms of BIP glue connectors) hard to achieve. Moreover, it is our objective to be able to model and verify both individual and joint behaviour, and we find this approach more appropriate for our domain. Another approach in [28] makes use of (global) behavioural types to focus on interactions and completely abstract away from local behaviour. However, we envision reconfigurable MASs as systems populating a surrounding environment. Thus, local behaviour that results from sensing the environment is very important to model and reason about. Indeed, in a previous work [2], we presented a concrete application in the realm of reconfigurable power distribution grids and showed how interaction protocols can be influenced by local behaviours.

Moreover, R-CHECK is uniquely distinguished from existing formalisms, whether they are attribute-based such as *AbC* or connector-based such as static BIP [13], due to run-time reconfiguration of interaction interfaces. In attribute-based communication, the interaction is based on value-passing broadcast, and thus there is no way to create a dedicated communication structure at run-time. For connector-based communication, the communication structure is static and cannot be changed. R-CHECK could be viewed as a generalisation of π-calculus [33] like reconfiguration in a multi-party settings. It is worth mentioning that R-CHECK is developed to be used as a specification language rather than a programming language. Thus, implementation issues are out of the scope of this article. For future work, we plan to investigate a possible specialised implementation for the Robot Operating System (ROS). This is in line with our goals to target collective behaviour in small-scale settings.

This would imply that a fully distributed semantics might not be required.

Several model-checking toolkits support specification languages that are designed to reason about concurrent systems and protocol design and thus allow to model processes that may interact with each other, usually via channel synchronisation. Examples include SPIN, mCRL [14], and CADP [24]. These toolkits are successful in reasoning about static coordination protocols, mainly related to fixed-structure systems like hardware and low-level communication protocols, but do not expand their coverage to MAS features. They also cannot handle infinite state systems, are usually tied to a limited choice of verification algorithms, and have limited support for interoperability; this last concern is partially mitigated by third-party projects such as LTSmin [29]. By contrast, the input language used by nuXmv [15] is designed at the semantic level of transition systems, making it an excellent candidate to serve as a backbone for special-purpose model-checking tools. Furthermore, this toolkit implements a large number of efficient algorithms for verification. These considerations led us to integrate R-CHECK with nuXmv.

Future works We plan to equip R-CHECK with a richer specification language that allows reasoning about the knowledge of agents and the dissemination of knowledge in distributed settings. For this purpose, we will investigate the possible integration of R-CHECK with MCMAS [31] to make use of the specialised symbolic algorithms that are introduced for knowledge reasoning. We also plan to integrate the partial order semantics of ReCiPe models that were introduced in [8].

Funding Open access funding provided by University of Gothenburg. This work was funded by the ERC consolidator grant D-SynMA (No. 772459) and the Swedish research council grants SynTM (No. 2020-03401) and VR project (No. 2020-04963).

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Abd Alrahman, Y., Piterman, N.: Modelling and verification of reconfigurable multi-agent systems. Auton. Agents Multi Agent Syst. **35**(2), 47 (2021). https://doi.org/10.1007/s10458-021-09521-x
2. Abd Alrahman, Y., Vieira, H.T.: A coordination protocol language for power grid operation control. J. Log. Algebraic Methods Program. **109** (2019). https://doi.org/10.1016/j.jlamp.2019.100487
3. Abd Alrahman, Y., De Nicola, R., Loreti, M.: A calculus for collective-adaptive systems and its behavioural theory. Inf. Comput. **268** (2019). https://doi.org/10.1016/j.ic.2019.104457
4. Abd Alrahman, Y., De Nicola, R., Loreti, M.: Programming interactions in collective adaptive systems by relying on attribute-based communication. Sci. Comput. Program. **192**, 102428 (2020). https://doi.org/10.1016/j.scico.2020.102428
5. Abd Alrahman, Y., Perelli, G., Piterman, N.: Reconfigurable interaction for MAS modelling. In: Seghrouchni, A.E.F., Sukthankar, G., An, B., Yorke-Smith, N. (eds.) Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems, AAMAS '20, Auckland, New Zealand, May 9-13, 2020, pp. 7–15. International Foundation for Autonomous Agents and Multiagent Systems (2020). https://doi.org/10.5555/3398761.3398768
6. Abd Alrahman, Y., Azzopardi, S., Piterman, N.: Model checking reconfigurable interacting systems. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Adaptation and Learning - 11th International Symposium, ISoLA 2022, Proceedings, Part III, Rhodes, Greece, October 22-30, 2022. Lecture Notes in Computer Science, vol. 13703, pp. 373–389. Springer, Berlin (2022). https://doi.org/10.1007/978-3-031-19759-8_23
7. Abd Alrahman, Y., Azzopardi, S., Piterman, N.: R-check: a model checker for verifying reconfigurable mas. In: Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems, AAMAS '22, pp. 1518–1520. International Foundation for Autonomous Agents and Multiagent Systems, Richland (2022). https://doi.org/10.5555/3535850.3536020
8. Abd Alrahman, Y., Martel, M., Piterman, N.: A PO characterisation of reconfiguration. In: Seidl, H., Liu, Z., Pasareanu, C.S. (eds.) Theoretical Aspects of Computing - ICTAC 2022 - 19th International Colloquium, Proceedings, Tbilisi, Georgia, September 27–29, 2022. Lecture Notes in Computer Science, vol. 13572, pp. 42–59. Springer, Berlin (2022). https://doi.org/10.1007/978-3-031-17715-6_5
9. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Reactive Systems: Modelling, Specification and Verification. Cambridge University Press, Cambridge (2007). https://doi.org/10.1017/CBO9780511814105
10. Alur, R., Henzinger, T.: Reactive modules. Form. Methods Syst. Des. **15**(1), 7–48 (1999)
11. Alur, R., Henzinger, T., Kupferman, O.: Alternating-time temporal logic. J. ACM **49**(5), 672–713 (2002). https://doi.org/10.1145/585265.585270
12. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: 3th International Conference on Software Engineering and Formal Methods (SEFM), pp. 3–12. IEEE, Pune (2006). https://doi.org/10.1109/SEFM.2006.27
13. Bliudze, S., Sifakis, J.: The algebra of connectors - structuring interaction in BIP. IEEE Trans. Comput. **57**(10), 1315–1330 (2008). https://doi.org/10.1109/TC.2008.26
14. Bunte, O., Groote, J.F., Keiren, J.J.A., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, W., Wijs, A., Willemse, T.A.C.: The mcrl2 toolset for analysing concurrent systems - improvements in expressivity and usability. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Proceedings, Part II, Prague, Czech Republic, April 6-11, 2019. Lecture Notes in Computer Science, vol. 11428, pp. 21–39. Springer (2019). https://doi.org/10.1007/978-3-030-17465-1_2
15. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification

- 24th International Conference, CAV 2012, Proceedings, Berkeley, CA, USA, July 7-13, 2012. Lecture Notes in Computer Science, vol. 7358, pp. 277–293. Springer, Berkeley (2012). https://doi.org/10.1007/978-3-642-31424-7_23
16. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) Computer Aided Verification, 14th International Conference, CAV 2002, Proceedings, Copenhagen, Denmark, July 27-31, 2002. Lecture Notes in Computer Science, vol. 2404, pp. 359–364. Springer (2002). https://doi.org/10.1007/3-540-45657-0_29
17. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
18. Cohen, P.R., Levesque, H.J.: Intention is choice with commitment. Artif. Intell. **42**(2–3), 213–261 (1990). https://doi.org/10.1016/0004-3702(90)90055-5
19. De Nicola, R., Latella, D., Lluch-Lafuente, A., Loreti, M., Margheri, A., Massink, M., Morichetta, A., Pugliese, R., Tiezzi, F., Vandin, A.: The SCEL language: design, implementation, verification. In: Wirsing, M., Hölzl, M.M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems - the ASCENS Approach. Lecture Notes in Computer Science, vol. 8998, pp. 3–71. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-16310-9_1
20. De Nicola, R., Maggi, A., Sifakis, J.: DReAM: dynamic reconfigurable architecture modeling. In: Margaria, T., Steffen, B. (eds.) 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA). LNCS, vol. 11246, pp. 13–31. Springer, Limassol (2018). https://doi.org/10.1007/978-3-030-03424-5_2
21. De Nicola, R., Duong, T., Inverso, O.: Verifying abc specifications via emulation. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Proceedings, Part II, Rhodes, Greece, October 20-30, 2020. Lecture Notes in Computer Science, vol. 12477, pp. 261–279. Springer (2020). https://doi.org/10.1007/978-3-030-61470-6_16
22. D'Ippolito, N., Fischbein, D., Chechik, M., Uchitel, S.: MTSA: the modal transition system analyser. In: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), L'Aquila, Italy, 15–19 September 2008, pp. 475–476. IEEE Comput. Soc., Los Alamitos (2008). https://doi.org/10.1109/ASE.2008.78
23. Fagin, R., Halpern, J., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press, Cambridge (1995)
24. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. Int. J. Softw. Tools Technol. Transf. **15**(2), 89–107 (2013). https://doi.org/10.1007/s10009-012-0244-z
25. Giannakopoulou, D., Magee, J.: Fluent model checking for event-based systems. In: Proceedings of the 9th European Software Engineering and 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 257–266. ACM, New York (2003)
26. Gutierrez, J., Harrenstein, P., Wooldridge, M.: From model checking to equilibrium checking: reactive modules for rational verification. Artif. Intell. **248**, 123–157 (2017). https://doi.org/10.1016/j.artint.2017.04.003
27. Hoare, C.A.R.: Communicating sequential processes. In: Jones, C.B., Misra, J. (eds.) Theories of Programming: The Life and Works of Tony Hoare, pp. 157–186. ACM / Morgan & Claypool, New York (2021). https://doi.org/10.1145/3477355.3477364
28. Inverso, O., Trubiani, C., Tuosto, E.: Abstractions for collective adaptive systems. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Proceedings, Part II, Rhodes, Greece, October 20-30, 2020. Lecture Notes in Computer Science, vol. 12477, pp. 243–260. Springer (2020). https://doi.org/10.1007/978-3-030-61470-6_15
29. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: Ltsmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, Proceedings, London, UK, April 11-18, 2015. Lecture Notes in Computer Science, vol. 9035, pp. 692–707. Springer, London (2015). https://doi.org/10.1007/978-3-662-46681-0_61
30. Legay, A., Lukina, A., Traonouez, L.M., Yang, J., Smolka, S.A., Grosu, R.: Statistical Model Checking, pp. 478–504. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-91908-9_23
31. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: an open-source model checker for the verification of multi-agent systems. Int. J. Softw. Tools Technol. Transf. **19**(1), 9–30 (2017)
32. Loreti, M., Hillston, J.: Modelling and analysis of collective adaptive systems with CARMA and its tools. In: Bernardo, M., De Nicola, R., Hillston, J. (eds.) Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems - 16th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2016, Bertinoro, Italy, June 20-24, 2016. Advanced Lectures, Lecture Notes in Computer Science, vol. 9700, pp. 83–119. Springer, Berlin (2016). https://doi.org/10.1007/978-3-319-34096-8_4
33. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes. Inform. and Comput. **100**(1), 1–40 (1992). https://doi.org/10.1016/0890-5401(92)90008-4
34. Nenzi, L., Bortolussi, L., Loreti, M.: jsstl - a tool to monitor spatio-temporal properties. In: Puliafito, A., Trivedi, K.S., Tuffin, B., Scarpa, M., Machida, F., Alonso, J. (eds.) 10th EAI International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS 2016, Taormina, Italy, 25th-28th Oct 2016. ACM, (2016). https://doi.org/10.4108/eai.25-10-2016.2266978
35. Nicola, R.D., Latella, D., Lluch-Lafuente, A., Loreti, M., Margheri, A., Massink, M., Morichetta, A., Pugliese, R., Tiezzi, F., Vandin, A.: The SCEL language: design, implementation, verification. In: Wirsing, M., Hölzl, M.M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems - the ASCENS Approach. Lecture Notes in Computer Science, vol. 8998, pp. 3–71. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-16310-9_1
36. Piterman, N., Pnueli, A.: Temporal logic and fair discrete systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 27–73. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-10575-8_2
37. Wooldridge, M.J.: An Introduction to MultiAgent Systems, 2nd edn. Wiley, New York (2009)
38. Zon, N., Gilmore, S., Hillston, J.: Rigorous graphical modelling of movement in collective adaptive systems. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Proceedings, Part I, Corfu, Greece, October 10-14, 2016. Lecture Notes in Computer Science, vol. 9952, pp. 674–688 (2016). https://doi.org/10.1007/978-3-319-47166-2_47

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