**R-CHECK: A Model Checker for Verifying Reconfigurable MAS**

**Yehia Abd Alrahman**  
University of Gothenburg  
Gothenburg, Sweden

**Shaun Azzopardi**  
University of Gothenburg  
Gothenburg, Sweden

**Nir Piterman**  
University of Gothenburg  
Gothenburg, Sweden

**ABSTRACT**

Reconfigurable multi-agent systems consist of a set of autonomous agents, with integrated interaction capabilities that feature opportunistic interaction. Agents seemingly reconfigure their interactions 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. We propose a model checker, named R-CHECK, to allow reasoning about these systems both from an individual- and a system-level. R-CHECK also permits reasoning about interaction protocols and joint missions. R-CHECK supports a high-level input language with symbolic semantics, and provides a modelling convenience for interaction features such as reconfiguration, coalition formation, self-organisation, etc.

**KEYWORDS**

Model-checking; Agent Theories and Models; Verification of Multi-Agent Systems

**ACM Reference Format:**  

**1 OVERVIEW OF R-CHECK**

R-CHECK\(^1\) accepts a high-level language that is based on the symbolic ReCiPe formalism [2, 3]. We will present the syntax of the R-CHECK language and informally describe its semantics. For a full exposition of the formal definition of R-CHECK and its usage through sizeable case studies, we refer the reader to [4].

An R-CHECK system is composed of several agent instances that communicate by sending messages over a given set of channels. Common predicates, which agents interpret (differently) over their local states, are used to decide for whom messages are intended.

We first introduce the module agent, its structure, and how to instantiate it; we introduce the syntax of its behaviour and how to create a system of agents. The module agent is reported in Fig. 1.

Each module has a name that identifies a specific type of behaviour. We permit creating multiple agent instances with the same type of behaviour. An agent has a local state “local” represented by a set of local variables \(V_f\), and interpretations of common predicates to interact with other agents anonymously. The initial state of an agent \(\text{init: } \theta_f\) is a predicate characterising the initial assignments to the agent local variables. receive-guard: \(\gamma'(V_f, Ch)\)

\(^1\)Find the associated toolkit repository here: https://github.com/dsynma/recipe.

![Figure 1: An agent module](image)

specifies the connectedness of the agent to channels given a current assignment to its local variables. The non-terminating behaviour of an agent is represented by \(\text{repeat: } P\), executing the process \(P\) indefinitely.

An agent module 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 \(\theta\). Here, we take the conjunction of \(\theta\) with the predicate in the 1st t section of the module “A” as the initial condition of this instance. We use the parallel composition operator \(\parallel\) to inductively define a system as in the following production rule:

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

That is, a system is either an agent or a parallel composition (with reconfigurable multicast and broadcast semantics as in [2, 3]) of agents. Agents interact by state-parametric message exchange.

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

\[
\text{(Process)}\quad P ::= \ P, P \ |\ P + P \ |\ \text{rep } P \ |\ C
\]

\[
\text{(Command)}\quad C ::= \ l : C \ |\ (\Phi) \ ch ! \ pi d U \ |\ (\Phi) \ ch ? U
\]

A process \(P\) is either a sequential composition of two processes \(P, P\), a non-deterministic choice between two processes \(P + P\), a loop \(\text{rep } P\), or a command \(C\). There are three types of commands: 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 checker to reason about syntactic elements as we will see later. A command of the form \((\Phi) \ ch ! \ pi d U\) corresponds to a message-send. The predicate \(\Phi\) is an assertion over the current local state of an agent, i.e., is a pre-condition that must hold before the transition can be taken. As the names suggest, \(ch, \pi\) and (respectively) \(d\) are the communication channel, the sender predicate (specifying the targeted receivers), and the assignment to data variables (i.e., the actual content of the message). Lastly, \(U\) is an update to local variables after taking the transition. We use “!” to distinguish send transitions. A command of the form \((\Phi) \ ch ? U\) corresponds to a message-receive. Differently from message-send, the predicate \(\Phi\) can also predicate on the received values from the incoming message, i.e., the assignment \(d\).
We show how to verify properties about agents both from individual and interaction protocols level by predating on message exchange rather than on atomic propositions. It should be noted that the transition labels in Fig. 2 are not mere labels, but rather predicates with truth values changing dynamically at run-time, introducing opportunistic interaction. For instance, we can reason about a client and its connection to the system as follows:

\[ G (\text{client1} \rightarrow \text{sReserve} \implies F (\text{client1} \rightarrow \text{sRelease}) \] (1)
\[ G (\text{client1} \rightarrow \text{sRequest} \implies F (\text{client1} \rightarrow \text{rConnect}) \] (2)

The liveness condition (1) specifies that the client does not hold a live lock on a shared link. Namely, the client releases the shared link eventually. The liveness condition (2) 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 in the following:

\[ G (\text{manager} \rightarrow \text{sForward} \implies X \text{machine1} \rightarrow \text{rForward}) \] (3)
\[ F (\text{client1} \rightarrow \text{sRelease} \land G (\text{client1} \rightarrow \text{rConnect})) \] (4)

In (3), we refer to synchronisation, i.e., the manager has to forward the request before the machine can receive it. We refer to reconfiguration in (4), i.e., eventually the client disconnects from the common link, and can never receive connections on that link.

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

\[
F (c(\text{client1} \rightarrow m\text{Link} \neq \text{empty}) \land F (\text{client2} \rightarrow m\text{Link} \neq \text{empty}) \land F (\text{client3} \rightarrow m\text{Link} \neq \text{empty})) \implies (F (\text{client1} \rightarrow \text{sSolve} \land \text{c(client2} \rightarrow \text{sSolve} \land \text{client3} \rightarrow \text{sSolve}))
\]

That is, every client will eventually receive a mobile link (i.e., its mLink \neq \text{empty}) where it will use this private link to get a VM, and eventually one client will initiate the termination of the mission by synchronising with the other clients to solve the joint problem.

\section{CONCLUDING REMARKS}

We introduced the R-CHECK model checking toolkit for verifying and simulating reconfigurable multi-agent system. R-CHECK is supported with a command line tool, a web editor with syntax highlighting and visualisation. We integrated R-CHECK with nuXmv to enable LTL symbolic (bounded) model checking. We showed that this specialised integration provides a powerful tool that permits verifying high-level features such as interaction protocols, joint missions, channel mobility, reconfiguration, self-organisation, etc.

R-CHECK combines the lessons learnt from communication models like \textit{AbC} [1, 5] and \textit{ReCiPe} [2, 3], and mainstreams model checkers like MCMAS [13] which is based on \textit{Interpreted Systems} [9], MTSA toolkit [8] (based on Hoare’s CSP calculus [11] and \textit{Fluent Linear Temporal logic} (FLTL) [10], SPIN [12] (for protocol design). Furthermore, R-CHECK strives for expressiveness while preserving minimality and simplicity.

\section{ACKNOWLEDGMENTS}

This work is 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).
REFERENCES


