RSC to the ReSCu: Automated Verification of Systems of Communicating Automata

Loïc Desgeorges, Loïc Germerie Guizouarn

RSC to the ReSCu: Automated Verification of Systems of Communicating Automata*

Loïc Desgeorges and Loïc Germerie Guizouarn

Université Côte d'Azur, CNRS, I3S, Sophia Antipolis, France
{loic.desgeorges,loic.germerie-guizouarn}@univ-cotedazur.fr

Abstract We present ReSCu, a model-checking tool for RSC (Realisable with Synchronous Communication) systems of communicating automata. Communicating automata are a formalism used to model communication protocols: each participant is represented by a finite state automaton, whose transitions are labelled by sending and receiving actions. In the general case, such automata exchanging messages asynchronously via FIFO or bag buffers are Turing-powerful, therefore most safety verification problems are undecidable. In RSC systems, the reception of a message may happen right after its send action. A lot of verification problems, e.g. reachability of a control state, are decidable for RSC systems. ReSCu checks whether a system is RSC, allowing to observe that a significant portion of protocols from the literature is RSC. This tool can also perform verification of safety properties for those systems, and is competitive in terms of time compared to non–RSC specific tools.

1 Introduction

Ensuring safety of communication protocols is admittedly a very important task. Systems of communicating automata (CA for short) are one of the formalisms modelling such protocols: each participant of the communication is represented by a finite state automaton, the transitions of which are labelled with actions, either to send or receive messages. Model-checking a system consists in verifying that it satisfies safety properties, e.g. whether an undesired configuration of control states is reachable. In this model, communications are asynchronous: messages are sent to unbounded buffers, waiting there to be received. The sender may immediately proceed with its subsequent actions. The main semantics for buffers are FIFO, for First In First Out, and bag. FIFO buffers behave like queues, messages are received in the same order as they were sent, whereas bag buffers allow receptions of messages in any order. Systems may be equipped with different structures of buffers named *communication architecture*. The most common ones being peer-to-peer, where there is one buffer per direction between each pair of participants, and mailbox, were each participant receives its messages from a single buffer.

* This work has been supported by the French government, through the EUR DS4H Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-17-EURE-0004.

From its asynchrony, comes a limitation of this model: buffers can encode the tape of a Turing Machine and therefore, deciding the reachability of a configuration of control states is undecidable [8]. Different strategies arose to circumvent this difficulty, the main ones being using semi-algorithms for verification, and restricting systems to classes in which verification problems become decidable.

The latter approach is the one we used in [11] and developed in [12]. Intuitively, a system is Realisable with Synchronous Communication (RSC for short) if all its executions can be reorganised to mimic a synchronous behaviour, where send and receive actions of the same message happen at the same time. Reachability of a regular set of configurations was shown to be decidable for RSC systems. Membership to the class of RSC systems is decidable as well, allowing to select the protocols on which the reachability algorithms can be used.

We present ReSCu (for **Re**alisable with **S**ynchronous **C**omm**u**nication), a model-checking tool for RSC systems of CA. This tool can answer whether a given system is RSC or not, and whether a specified bad configuration is reachable. ReSCu works on systems with any communication architecture (not restricted to peer-to-peer or mailbox) and either with bag or FIFO buffers.

Outline. After a discussion about related works, we will begin with some intuition about CA in Section 2. In Section 3, we present the tool itself, how it is implemented and how it can be used. Before concluding (Section 5), we will present some results and benchmarks we obtained with our tool in Section 4.

Related works. The closest tool to ReSCu is McScM [14]. It takes a description of a system and a set of bad configurations (defined as QDDs [5]), and checks whether a bad configuration is reachable. This tool implements various model-checking approaches, based on abstract interpretation. It is not limited to systems of a specific class. Contrary to ReSCu, most of these approaches are semi-algorithms and need a time-out to be set arbitrarily. However, the strength of McScM is the multiplicity of model-checking engines it provides, increasing the likelihood of a conclusive result for any system. We use its description language as a way to input systems in ReSCu.

The notion of *stability*, introduced in [4], is close to RSC. A system is k-stable if its behaviour with any bound $k' > k$ is equivalent (with several notions of equivalence possible) to its behaviour with a bound k. Model-checking can be performed with bounded buffers for stable systems. Stability was shown to be undecidable for FIFO systems in [3], but decidable with bag buffers (for a specific notion of equivalence) [2]. The authors of [2, 3] developed STABC: a tool using semi-algorithms to check k-stability of systems. Contrary to ReSCu, it does not perform verification of safety properties, but provides only membership results.

Lange and Yoshida proposed another tool: KMC [19], for k Multiparty Compatibility. It checks whether a system could have been obtained by projection of a global type, relying on the theory of Multiparty Session Types [15] (another way to model distributed systems). If a system is k-MC, various safety properties are ensured, and it is not necessary to specify them as it is for McScM or ReSCu.

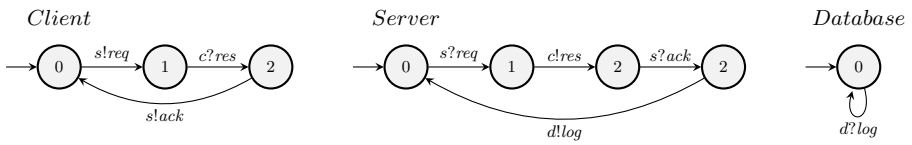

Figure 1: Protocol from Example 1

2 Communicating automata

We begin with a small example of protocol, borrowed from [3].

Example 1 (Communication protocol). We will consider a generic client/server protocol, enhanced with a database logging activity. In this protocol, the client may send a request to the server, and when it receives a result for this request, it sends an acknowledgement back to the server. The server waits for a request, and upon receiving it, it sends a result to the client. After that, it waits for an acknowledgement from the client and sends a logging message to the database. Those behaviours can be repeated indefinitely. □

Figure 1 is a graphical representation of the system of CA modelling the protocol from Example 1. Each participant is represented by an automaton, which can change states by executing the actions labelling its transitions. An action $i!v$ means that message v is sent in buffer i, and $i?v$ that v is received from buffer i. In this system each participant receives all its messages in a single FIFO buffer (mailbox). Informally, a *configuration* is the product of the control states of each participant, paired with the content of the buffers. A configuration is *reachable* if a sequence of actions of the system can lead to it. We focus on safety properties that can be expressed as a regular language of 'bad' configurations of a system. We say that such a safety property is satisfied if no configuration of the language is reachable in the system.

Example 2 (Safety specifications). In Example 1, the configuration where the server is in state 1, and the client in state 0 is a bad configuration: it means those two participants are not at the same step of the protocol any more. Both the server and the client are not ready to receive the messages they are about to send each other. In this tiny example, it is easy to see that such a configuration is not reachable, but on bigger systems an automatic verification may be useful to ensure such properties. Another example: a set of bad configurations is formed by the ones where the server is in state 0, and the first message in buffer s is not req, preventing any further reception to happen for this participant (indeed, remember we use FIFO buffers, only the first message of a buffer may be received). □

Intuitively, a system of CA is RSC if send actions and their respective reception can happen at the same time: there is no need for another action to be performed between sending a message and receiving it. The work in [12] provides formal definitions of CA and RSC systems, as well as algorithms for deciding membership to the class of RSC systems and reachability of a configuration.

```
scm client_server_database :

nb_channels = 3;
parameters: int req; int res;
int log; int ack;

automaton server:
initial: 0
state 0:
to 1: when true, 0 ? req;
state 1:
to 2: when true, 1 ! res;
state 2:
to 3: when true, 0 ? ack;
state 3:
to 0: when true, 2 ! log;

automaton database:
initial: 0
state 0:
to 0: when true, 2 ? log;
```

```
automaton client:
initial: 0
state 0:
to 1: when true, 0 ! req;
state 1:
to 2: when true, 1 ? res;
state 2:
to 0: when true, 0 ! ack;

bad_states:
(automaton client: in 0: true
 automaton server: in 1: true)

(automaton server: in 0: true with
 (log|res|ack).(req|res|log|ack)^*.#.
 (req|res|log|ack)^*.#.
 (req|res|log|ack)^*)
```

Figure 2: SCM representation of Example 1

3 ReSCu

ReSCu is a tool using the properties of RSC systems to perform model-checking. It is an OCaml implementation of the algorithms in [12]. While working on this implementation, we discovered a bug in the membership algorithm; we provide in Appendix A the fixed algorithms, generalised to take into account bag buffers. ReSCu provides a command line interface that takes a file describing a system of CA and its safety specifications, and outputs whether this system is RSC, and whether a bad configuration is reachable or not. If a bad configuration is reachable, ReSCu can display the execution leading to the safety counterexample. Similarly, if non-RSC executions are possible, one of them may be displayed. This tool is available at [10].

SCM description language. We chose, as an input format, the SCM language used in [14]. This allowed to rely on the parser that was already available thanks to the developers of McScM, and to compare easily ReSCu with this tool. Figure 2 shows the SCM description of the system in Example 1. The set of messages is declared after the keyword `parameters`, and the number of buffers after the keyword `nb_channels` ('channel' is the name used for buffers in SCM). An automaton is declared as a list of states, each of them containing a (possibly empty) list of transitions. SCM allows specification of model features we did not take into consideration, hence the '`when true`' in the transitions, or the types of each message. Bad configurations are declared after the keyword '`bad_states`', each one of them being a list of control states and an optional regular expression describing buffer contents. The bad states of this listing correspond to the ones in Example 2.

Usage. The command line utility allows to check both membership and safety of a system: `rescu -isrsc <system>` checks whether the system described in the

Protocol	$\|\mathbb{P}\|$	S	T	RSC	t_{rsc}	k-MC	t_{kmc}
SMTP [16, 21]	2	64	108	Yes	17	Yes	34
HTTP [17, 21]	2	12	48	Yes	17	Yes	28
Elevator [6]	3	13	23	No	7	Yes	41
Commit protocol [6]	4	12	12	Yes	4	Yes	15
Travel agency [21]	3	17	20	Yes	8	Yes	15
SH [21]	3	22	30	Yes	18	Yes	33

(a) Comparison with KMC

Protocol	$\|\mathbb{P}\|$	S	T	RSC	t_{rsc}	k	t_{stabc}
Estelle specification [18]	2	7	9	No	5	max	82,625
FTP transfer [7]	3	20	17	Yes	6	4	89,465
SQL server [22]	4	33	38	Yes	13	3	90,553
SSH [20]	4	27	28	Yes	7	2	43,855
Bug report repository [13]	4	11	11	Yes	4	max	134,796
Restaurant service [1]	3	16	16	No	5	2	52,793

(b) Comparison with STABC, using FIFO buffers and 'strong equivalence'. max means the arbitrary limit for k, set at 10, was reached.

Table 1: Membership results of ReSCu compared with KMC and STABC. $|\mathbb{P}|$ is the number of participants, S the number of states, and T the number of transitions. t_{rsc}, t_{kmc} and t_{stabc} denote the time (in ms) of computation of ReSCu, KMC and STABC.

SCM file `<system>` is RSC, and `rescu -mc <system>` checks that no bad configuration is reachable. The two options can be combined in one call to ReSCu. Option `-bag` specifies that all buffers should be considered as bag buffers. In this case bad specifications including a description of the buffer contents are not accepted. For convenience while testing, we included a feature allowing to output a DOT representation of an SCM file. A video demonstrating the use of ReSCu is available at [9].

Implementation choices. McScM was designed as a framework, allowing addition of model-checking engines as modules. We opted for a stand-alone tool as the interface with McScM is way more involved than what is required for RSC algorithms. In addition, McScM is no longer maintained, and in its current state it is not possible to compile it with a modern version of OCaml.

4 Results

We used ReSCu on the set of examples provided with McScM, and we ported examples of systems available with STABC [3] and KMC [19]. This allowed to test our tool on a lot of systems, some of which modelling actual protocols.

Proportion of RSC systems in the wild. We used ReSCu to check the existence of RSC systems among examples from the literature. Using FIFO buffers, 30% of the systems from [14], 60% of those from [19] and 38% of those from [3] are RSC. Using bag buffers, the results are respectively 12%, 41% and 11%. These figures are to be interpreted carefully however, as the examples coming from KMC and STABC are not all random examples. Examples of systems from KMC are CSA, for *communicating session automata*, which is a class of systems where there cannot be sending and receiving transitions leaving the same state. Some systems where even (slightly) modified to become CSA. To provide a more realistic overview of the importance of RSC systems in the literature, we show in Table 1 some membership results for interesting protocols that were featured in [19] and [3]. It shows a comparison of the results of ReSCu on one hand, and the results we reproduced with their respective tools on the other hand. The k

value provided by STABC is a buffer bound that may be applied to the system without restricting its behaviour. An extended version of those tables is available in Appendix B.

Performance of our tool. We ran both our tool and McScM on several RSC examples from McScM, KMC and STABC, and compared the model-checking time. For the ported examples, we had to design some specifications, as the tools those systems came from focused only on membership to a class. The bad configurations we added are similar to the second one of Figure 2: they enforce that, for a specific control state of a participant, no configuration where the first message of the buffer cannot be received is reached. For reference, we ran our testing on a laptop with an Intel Core i5-8250U CPU at 1.60GHz, equipped with 16Gb of RAM.

Protocol	*ReSCu*	*absint*	*armc*	*cegar*	*lart*
ring	**137**	(19,708)	T_{max}	382	1,928
NonRegular	**4**	60	T_{max}	13	10
pop3	**33**	719	2,143	6,759	T_{max}
Nested	**4**	5	11	320	2045
con_disc_reg	**4**	(21)	7	9	**4**
tcp_error*	**4**	(107)	26	66	10
http-fsm	**7**	44	T_{max}	T_{max}	T_{max}
smtp	**84**	236	241	174	173
FTP	51	**29**	54	61	82
SSH	207	574	368	**188**	910

Table 2: Model-checking time (in ms) of ReSCu and McScM. Figures in brackets correspond to inconclusive verification.

Table 2 presents the times of computation of the different algorithms, averaged over 3 runs. The shortest time for each protocol is highlighted. The three horizontal sections of the table correspond to the origin of the examples: McScM, KMC and STABC, in that order. The runs that reached the time limit of 2 minutes are marked T_{max}. The columns for `cegar` and `lart` present the best time of the four variants of these algorithms.

As an example, we detail the results for a protocol: `ring`, a token passing protocol in a ring with four peers. The first algorithm, `absint`, did not provide a conclusive answer, and ran for about 19 seconds. The second one, `armc`, reached the time limit we set at 2 minutes without finishing. The next algorithms have four variants each, and even if `cegar` is the fastest in this example, one of its variant times out. Two of the variants of `lart` time out as well.

The protocol `tcp_error`* is a simplified version of TCP, intentionally modified to be erroneous. It is not RSC, but we included it as ReSCu can still find one of its bad configurations. Even though ReSCu cannot certify that a non-RSC protocol is safe, it can still help finding bugs quickly.

The rightmost column in Tables 1a and 1b gives an overview of the performance of the membership algorithm, compared to KMC and STABC respectively. Note that KMC checks the safety of a protocol, while knowing if a given system is RSC merely allows to know if our model-checking algorithm is suitable for it.

5 Conclusion

We presented ReSCu, a tool relying on the properties of RSC systems of communicating automata to verify safety of communication protocols. Through extensive testing and comparison with other tools, ReSCu proved to be performant,

and allowed to notice that a significant portion of actual protocols from the literature are indeed RSC.

This tool has some limitations however: some systems are not RSC, and ReSCu cannot certify safety of those. Another drawback is that while other tools can check various safety properties taking only the description of the protocol, we need the users to define correctly the safety properties they want to check. While our current setting allows for some flexibility, generating bad configurations automatically for properties like unspecified reception, or progress (see [12]), could be an interesting improvement of ReSCu, and is left as future work.

Acknowledgements. We would like to thank all the COORDINATION reviewers for their comments that greatly improved the present paper.

References

[1] Wil M. P. van der Aalst, Arjan J Mooij, Christian Stahl and Karsten Wolf. 'Service interaction: Patterns, formalization, and analysis'. In: *Formal Methods for Web Services, SFM, Advanced Lectures 9* (2009). Publisher: Springer, pp. 42–88.

[2] Lakhdar Akroun and Gwen Salaün. 'Automated verification of automata communicating via FIFO and bag buffers'. In: *Formal Methods Syst. Des.* 52.3 (2018), pp. 260–276. DOI: 10.1007/s10703-017-0285-8.

[3] Lakhdar Akroun, Gwen Salaün and Lina Ye. 'Automated Analysis of Asynchronously Communicating Systems'. In: *Model Checking Software - 23rd International Symposium, SPIN, Proceedings*. Vol. 9641. Lecture Notes in Computer Science. Springer, 2016, pp. 1–18. DOI: 10.1007/978-3-319-32582-8_1.

[4] Samik Basu and Tevfik Bultan. 'Automatic verification of interactions in asynchronous systems with unbounded buffers'. In: *ACM/IEEE International Conference on Automated Software Engineering, ASE*. ACM, 2014, pp. 743–754. DOI: 10.1145/2642937.2643016.

[5] Bernard Boigelot and Patrice Godefroid. 'Symbolic Verification of Communication Protocols with Infinite State Spaces Using QDDs (Extended Abstract)'. In: *Computer Aided Verification, 8th International Conference, CAV, Proceedings*. Vol. 1102. Lecture Notes in Computer Science. Springer, 1996, pp. 1–12. DOI: 10.1007/3-540-61474-5_53.

[6] Ahmed Bouajjani, Constantin Enea, Kailiang Ji and Shaz Qadeer. 'On the Completeness of Verifying Message Passing Programs Under Bounded Asynchrony'. In: *Computer Aided Verification - 30th International Conference, CAV, Held as Part of the Federated Logic Conference, FloC, Proceedings, Part II*. Vol. 10982. Lecture Notes in Computer Science. Springer, 2018, pp. 372–391. DOI: 10.1007/978-3-319-96142-2_23.

[7] Andrea Bracciali, Antonio Brogi and Carlos Canal. 'A formal approach to component adaptation'. In: *Journal of Systems and Software* 74.1 (2005). Publisher: Elsevier, pp. 45–54.

[8] Daniel Brand and Pitro Zafiropulo. 'On Communicating Finite-State Machines'. In: *Journal of the ACM* 30.2 (1983), pp. 323–342. DOI: 10.1145/322374.322380.

[9] Loïc Desgeorges and Loïc Germerie Guizouarn. *Demonstration video of ReSCu*. https://seafile.celazur.fr/f/bfa8e1380ce540f5bddb/?dl=1.

[10] Loïc Desgeorges and Loïc Germerie Guizouarn. *ReSCu archive*. https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://src.koda.cnrs.fr/loic.germerie.guizouarn/rescu.

[11] Cinzia Di Giusto, Loïc Germerie Guizouarn and Étienne Lozes. 'Towards Generalised Half-Duplex Systems'. In: *14th Interaction and Concurrency Experience, ICE, Proceedings*. Vol. 347. EPTCS. 2021, pp. 22–37. DOI: 10.4204/EPTCS.347.2.

[12] Cinzia Di Giusto, Loïc Germerie Guizouarn and Etienne Lozes. 'Multiparty half-duplex systems and synchronous communications'. In: *Journal of Logical and Algebraic Methods in Programming* 131 (2023), p. 100843. ISSN: 2352-2208. DOI: 10.1016/j.jlamp.2022.100843.

[13] Gregor Gößler and Gwen Salaün. 'Realizability of Choreographies for Services Interacting Asynchronously'. In: *Formal Aspects of Component Software - 8th International Symposium, FACS, Revised Selected Papers*. Vol. 7253. Lecture Notes in Computer Science. Springer, 2011, pp. 151–167. DOI: 10.1007/978-3-642-35743-5_10.

[14] Alexander Heußner, Tristan Le Gall and Grégoire Sutre. 'McScM: A General Framework for the Verification of Communicating Machines'. In: *Tools and Algorithms for the Construction and Analysis of Systems, TACAS, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS, Proceedings*. Vol. 7214. Lecture Notes in Computer Science. Springer, 2012, pp. 478–484. DOI: 10.1007/978-3-642-28756-5_34.

[15] Kohei Honda, Nobuko Yoshida and Marco Carbone. 'Multiparty asynchronous session types'. In: *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, Proceedings*. ACM, 2008, pp. 273–284. DOI: 10.1145/1328438.1328472.

[16] Raymond Hu. 'Distributed programming using Java APIs generated from session types'. In: *Behavioural Types: from Theory to Tools* (2017). Publisher: River Publishers, pp. 287–308.

[17] Raymond Hu and Nobuko Yoshida. 'Hybrid session verification through endpoint API generation'. In: *Fundamental Approaches to Software Engineering, FASE, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS, Proceedings 19*. Springer, 2016, pp. 401–418.

[18] Thierry Jéron and Claude Jard. 'Testing for unboundedness of fifo channels'. In: *Theoretical Computer Science* 113.1 (1993). Publisher: Elsevier, pp. 93–117.

[19] Julien Lange and Nobuko Yoshida. 'Verifying Asynchronous Interactions via Communicating Session Automata'. In: *Computer Aided Verification - 31st International Conference, CAV, Proceedings, Part I*. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 97–117. DOI: 10.1007/978-3-030-25540-4_6.

[20] José Antonio Martín and Ernesto Pimentel. 'Contracts for security adaptation'. In: *The Journal of Logic and Algebraic Programming* 80.3 (2011). Publisher: Elsevier, pp. 154–179.

[21] Rumyana Neykova, Raymond Hu, Nobuko Yoshida and Fahd Abdeljallal. 'A session type provider: compile-time API generation of distributed protocols with refinements in F#'. In: *27th International Conference on Compiler Construction, CC, Proceedings*. ACM, 2018, pp. 128–138. DOI: 10.1145/3178372.3179495.

[22] Pascal Poizat and Gwen Salaün. 'Adaptation of open component-based systems'. In: *Formal Methods for Open Object-Based Distributed Systems: 9th IFIP WG 6.1 International Conference, FMOODS, Proceedings 9*. Springer, 2007, pp. 141–156.

A Theoretical background

In this section, we present a condensed version of some theoretical of [12]. We introduce two modifications: the first one is that we generalise our previous work to systems of CA with bag buffers, and the second one is that we fix a small bug that existed in the definition of $\mathcal{A}_{bv}$ (defined in Section A.3).

A.1 Preliminaries

For a finite set S, S^* denotes the set of finite words over S, $w \cdot w'$ denotes the concatenation of words w and w', $|w|$ denotes the length of word w, and ε denotes the empty word. We write $\mathcal{L}(\mathcal{A})$ for the language accepted by automaton $\mathcal{A}$. For two sets S and I, we write $\mathbf{b}$ for an element of S^I, and b_i for the i-th component of $\mathbf{b} = (b_i)_{i \in I}$.

The set of all participants of a protocol is denoted $\mathbb{P}$. For a participant $p \in \mathbb{P}$, the *communicating automaton* $\mathcal{A}_p$ is the tuple $(L_p, \mathbb{V}_p, I_p^F, I_p^B, \mathsf{Act}_p, \delta_p, l_p^0)$ where:

- L_p is a finite set of control states,
- $\mathbb{V}_p$ is a finite set of *messages*,
- $I_p = I_p^F \cup I_p^B$ with $I_p^B \cap I_p^F = \varnothing$ is a finite set of *buffer identifiers* where I_p^B (respectively I_p^F) is the subset of bag (respectively FIFO) buffer identifiers,
- $\mathsf{Act}_p \subseteq I_p \times \{!^p, ?^p\} \times \mathbb{V}_p$ is a finite set of *actions*,
- $\delta_p \subseteq L_p \times \mathsf{Act}_p \times L_p$ is a finite set of *transitions*, and
- l_p^0 is the initial control state.

An action (denoted a) can be a send action: $i!^p v$, meaning 'process p sends message v in buffer i', or a reception: $i?^p v$ meaning 'process p receives message v from buffer i'. To ease readability, the process is omitted when the context allows it. For $a = i \dagger v$ with $\dagger \in \{!, ?\}$, $\mathsf{buffer}(a) = i$.

A system, denoted by $\mathfrak{S}$, is a family of communicating automata, one per participant $p \in \mathbb{P}$. For a system $\mathfrak{S} = (\mathcal{A}_p)_{p \in \mathbb{P}}$:

- $L_{\mathfrak{S}} = \prod_{p \in \mathbb{P}} L_p$ is the set of global control states of the system: for $\mathbf{l} \in L_{\mathfrak{S}}$, $\mathbf{l} = (l_p)_{p \in \mathbb{P}}$ is a vector of control states, where for each participant p, l_p is a control state of the automaton representing p;
- $\mathbb{V}_{\mathfrak{S}} = \cup_{p \in \mathbb{P}} \mathbb{V}_p$ is the set of messages;
- $I_{\mathfrak{S}} = I_{\mathfrak{S}}^F \cup I_{\mathfrak{S}}^B$ is the set of buffer identifiers, where $I_{\mathfrak{S}}^F = \cup_{p \in \mathbb{P}} I_p^F$ is the set of FIFO buffers identifiers and $I_{\mathfrak{S}}^B = \cup_{p \in \mathbb{P}} I_p^B$ is the set of bag buffers identifiers;
- $\mathsf{Act}_{\mathfrak{S}} = \cup_{p \in \mathbb{P}} \mathsf{Act}_p$ is the set of actions;
- $\delta_{\mathfrak{S}} = \left\{ (\mathbf{l}, a, \mathbf{l}') \mid \exists p \in \mathbb{P}, (l_p, a, l'_p) \in \delta_p, \forall q \neq p, l_q = l'_q \right\}$.

For an action $a \in \mathsf{Act}_{\mathfrak{S}}$, $\mathsf{process}(a)$ is the unique $p \in \mathbb{P}$ such that $a \in \mathsf{Act}_p$. A *configuration* (denoted γ) is a pair $(\mathbf{l}, \mathbf{b})$ where $\mathbf{l}$ is a global control states and $\mathbf{b} = (b_i)_{i \in I_{\mathfrak{S}}}$ is a vector of buffers, each b_i being the concatenation of the messages contained in the buffer i. The initial configuration of a system is $\gamma_0 = (\mathbf{l}_0, \mathbf{b}_0)$, with $\mathbf{l}_0 = (l_p^0)_{p \in \mathbb{P}}$ and $\mathbf{b}_0 = (\varepsilon)_{i \in I_{\mathfrak{S}}}$.

A *transition* of $\mathfrak{S}$ is a tuple (γ, a, γ'), often written $\gamma \xrightarrow[\mathfrak{S}]{a} \gamma'$, where $\gamma = (\mathbf{l}, \mathbf{b})$ and $\gamma' = (\mathbf{l}', \mathbf{b}')$ are two configurations, a is an action, and the following holds:

- $(\mathbf{l}, a, \mathbf{l}') \in \delta_{\mathfrak{S}}$
- if $a = i!^p v$, then $b'_i = v \cdot b_i$, and for all $j \in I_{\mathfrak{S}}, j \neq i, b_j = b'_j$
- if $a = i?^p v$, then for all $j \in I_{\mathfrak{S}}, j \neq i, b_j = b'_j$ and
  - if $i \in I^F_{\mathfrak{S}}$ then $b_i = v \cdot b'_i$
  - if $i \in I^B_{\mathfrak{S}}$ then $\exists w, w' \in \mathbb{V}^*_{\mathfrak{S}}$, $b_i = w \cdot v \cdot w'$, and $b'_i = w \cdot w'$.

An *execution* of a system $\mathfrak{S}$ is a word on $\mathsf{Act}^*_{\mathfrak{S}}$. We say that an execution $e = a_1 \cdot a_2 \cdot \ldots \cdot a_n$ is *feasible* in $\mathfrak{S}$ if there exists a sequence of configurations $\gamma_1, \gamma_2, \ldots, \gamma_n$ such that for all $i \in \{1, \ldots, n\}$, $\gamma_{i-1} \xrightarrow[\mathfrak{S}]{a_i} \gamma_i$. We write $\gamma_0 \xRightarrow[\mathfrak{S}]{e} \gamma_n$ for $\gamma_0 \xrightarrow[\mathfrak{S}]{a_1} \gamma_1 \xrightarrow[\mathfrak{S}]{a_2} \cdots \xrightarrow[\mathfrak{S}]{a_n} \gamma_n$, and by abuse of notation we write $a \in e$ if $\exists i \in \{1, \ldots, n\}, a = a_i$. The set of all feasible executions of $\mathfrak{S}$ is denoted $\mathsf{executions}(\mathfrak{S})$. A configuration γ of $\mathfrak{S}$ is reachable if there exists an execution $e \in \mathsf{Act}^*_{\mathfrak{S}}$ such that $\gamma_0 \xRightarrow[\mathfrak{S}]{e} \gamma$. The set of all reachable configurations of $\mathfrak{S}$ is denoted $RS(\mathfrak{S})$.

Given an execution $e = a_1 \cdot \ldots \cdot a_n$, we say that $\{a_j, a_{j'}\} \subseteq \{a_1, \ldots, a_n\}$ with $j < j'$ is a *matching pair* if there exist i, v, such that:

- $a_j = i!v$,
- $a_{j'} = i?v$,
- and
  - if $i \in I^F_{\mathfrak{S}}$ then $\exists k$ such that a_j (respectively $a_{j'}$) is the k-th send action (respectively reception) on i in e,
  - else $\exists k$ such that a_j (respectively $a_{j'}$) is the k-th send action (respectively reception) of message v on i in e.

For bag buffers, we say that when the same message is sent several times to a buffer, receptions of this message match the send actions in their order. A send action a_j is *unmatched* in e if there is no j' such that $\{a_j, a_{j'}\}$ is a matching pair. A *communication* (denoted c) is either a matching pair, or $\{a\}$ with a an unmatched send.

We say that two actions *commute* if they do not form a matching pair, and if they are not actions of the same type on a FIFO buffer. For an execution $e = a_1 \cdot \ldots \cdot a_n$, we say that $a_j \prec_e a_{j'}$, with $\{j, j'\} \subseteq \{1, \ldots, n\}$ if $j < j'$ and a_j does not commute with $a_{j'}$. Intuitively, $\prec_e$ represents causal dependencies between actions of an execution. Two executions $e = a_1 \cdot \ldots \cdot a_n$ and $e' = a'_1 \cdot \ldots \cdot a'_n$ are causally equivalent (denoted by $e \sim e'$) if there is a permutation σ of $\{1, \ldots, n\}$ such that:

- for all $i \in \{1, \ldots, n\}, a'_{\sigma(i)} = a_i$, and
- for all $j, j' \in \{1, \ldots, n\}, a_j \prec_e a_{j'}$ if and only if $a'_{\sigma(j)} \prec_{e'} a'_{\sigma(j')}$.

A property P is a function from a system to a set of configurations. For a system $\mathfrak{S}$, $P(\mathfrak{S})$ is the set of configurations of $\mathfrak{S}$ satisfying the property. A system is P safe if $P(\mathfrak{S}) \cap RS(\mathfrak{S}) = \varnothing$. The idea behind P safety of a system is to describe the configurations that should *not* be reached, and to check whether all of them are indeed unreachable. We say that a property P is regular if, for all $\mathfrak{S}$, there exists a finite state automaton recognising a set of words that encodes configurations in $P(\mathfrak{S})$.

A.2 RSC systems

An execution is RSC if all its receptions are immediately preceded by the matching send action. Such an execution is a sequence of unmatched send actions and matching pairs. A system is RSC if all its executions are causally equivalent to an RSC execution.

The set of RSC executions of a system is regular. For a system $\mathfrak{S}$, we can compute an automaton $\mathcal{A}_{rsc}(\mathfrak{S})$ accepting all RSC executions feasible in $\mathfrak{S}$. Formally, for $\mathfrak{S}$ a system, let $\Sigma_{\mathfrak{S}} = \{i!?v \mid i!v \in \mathsf{Act}_{\mathfrak{S}}, i?v \in \mathsf{Act}_{\mathfrak{S}}\} \cup \{i!v \mid i!v \in \mathsf{Act}_{\mathfrak{S}}\}$ the set of all communications, where $i!?v$ stands for the communication grouping the send and reception of v in buffer i. Let $\mathcal{A}_{rsc}(\mathfrak{S}) = (L_{rsc}, \delta_{rsc}, l^0_{rsc}, L^f_{rsc})$ be the non-deterministic finite state automaton over $\Sigma_{\mathfrak{S}}$ with $L_{rsc} = L_{\mathfrak{S}} \times 2^{I^F_{\mathfrak{S}}}$ its set of control states, $l^0_{rsc} = (\mathbf{l}_0, \varnothing)$ its initial state, and $L^f_{rsc} = L_{rsc}$ its set of accepting states (all states are accepting). For $c \in \Sigma_{\mathfrak{S}}$, $(\mathbf{l}, S) \in L_{rsc}$, $(\mathbf{l}', S') \in L_{rsc}$, $((\mathbf{l}, S), c, (\mathbf{l}', S')) \in \delta_{rsc}$ if:

- $(\mathbf{l}, \mathbf{b}) \xRightarrow[\mathfrak{S}]{c} (\mathbf{l}', \mathbf{b}')$ for some $\mathbf{b}, \mathbf{b}'$ such that for all $i \in I^F_{\mathfrak{S}}$, $b_i \neq \varepsilon$ iff $i \in S$, and $b'_i \neq \varepsilon$ iff $i \in S'$, and
- if $c = i!?v$, $i \notin S$.

A.3 Membership

A *bordeline violation* is a minimal non-RSC execution. An execution e is a borderline violation if it is not causally equivalent to an RSC execution, and it is of the form $e = e' \cdot i?v$ with e' RSC. By [12, Lemma 9], a system $\mathfrak{S}$ is RSC if and only if $\mathsf{executions}(\mathfrak{S})$ contains no borderline violation. The set of borderline violations of a system is regular: we define now $\mathcal{A}_{bv}$, an automaton recognising all borderline violations of a system.

Formally, for a system $\mathfrak{S}$, let $\Sigma^?_{\mathfrak{S}} = \{i?v \mid i \in I_{\mathfrak{S}}, v \in \mathbb{V}_{\mathfrak{S}}\}$ the alphabet of all possible receptions of $\mathfrak{S}$, and $\mathcal{A}_{bv} = \left(L_{bv}, \delta_{bv}, \left(l^0_{bv}, \varnothing\right), \{l^1_{bv}\}\right)$ the non-deterministic finite state automaton over $\Sigma_{\mathfrak{S}} \cup \Sigma^?_{\mathfrak{S}}$ such that $L_{bv} = \left\{l^0_{bv} \times 2^{|I^F_{\mathfrak{S}}|}, l^1_{bv}\right\} \cup (I_{\mathfrak{S}} \times \mathbb{V}_{\mathfrak{S}} \times \Sigma_{\mathfrak{S}} \times \{0,1\})$, and for all $c, c' \in \Sigma_{\mathfrak{S}}$, and for all $i \in I_{\mathfrak{S}}, v \in \mathbb{V}_{\mathfrak{S}}$:

1. $((l^0_{bv}, S), i!?v, (l^0_{bv}, S)) \in \delta_{bv}$ if $i \notin S$
2. $((l^0_{bv}, S), i!v, (l^0_{bv}, S')) \in \delta_{bv}$ if $i \in I^F_{\mathfrak{S}}$ and $S' = S \cup \{i\}$, or $S = S'$
3. $((l^0_{bv}, S), i!v, (i, v, i!v, 0) \in \delta_{bv}$ if $i \notin S$ or $i \in I^B_{\mathfrak{S}}$
4. $((i, v, c, 0), c', (i, v, c, 0)) \in \delta_{bv}$ if $i \in I^F_{\mathfrak{S}}$ or $c' \neq i!v$
5. $((i, v, c, 0), c', (i, v, c', 1)) \in \delta_{bv}$ if either $\mathsf{process}(c') \cap \mathsf{process}(c) \neq \varnothing$, or $\mathsf{buffer}(c) \in I^F_{\mathfrak{S}}$ and $\mathsf{buffer}(c) = \mathsf{buffer}(c')$
6. $((i, v, c, 1), c', (i, v, c, 1)) \in \delta_{bv}$ if $i \in I^F_{\mathfrak{S}}$ or $c' \neq i!v$
7. $((i, v, c, 1), c', (i, v, c', 1)) \in \delta_{bv}$ if either $\mathsf{process}(c') \cap \mathsf{process}(c) \neq \varnothing$, or $\mathsf{buffer}(c) \in I^F_{\mathfrak{S}}$ and $\mathsf{buffer}(c) = \mathsf{buffer}(c')$
8. $((i, v, c, 1), i?v, l^1_{bv}) \in \delta_{bv}$ if $\mathsf{process}(c) = \mathsf{process}(i?v)$, or c is a matching pair and $\mathsf{buffer}(c) \in I^F_{\mathfrak{S}}$ and $\mathsf{buffer}(c) = i$.

Theorem 1. *Whether a system of communicating automata is RSC is decidable.*

Proof. Let $\mathfrak{S}$ be a system of communicating automata, and let $\mathcal{L}$ be the intersection of $\mathcal{L}(\mathcal{A}_{bv}(\mathfrak{S}))$ on one hand, and $\mathcal{L}(\mathcal{A}_{rsc}(\mathfrak{S})) \cdot \Sigma_{\mathfrak{S}}^{?}$ on the other hand, then $\mathcal{L} = \varnothing$ iff $\mathfrak{S}$ is RSC. □

A.4 Reachability

Regular properties describe a set of control states, and a content for the different buffers. The contents of the buffers are described as set of words, specifying an order for the messages. We opted for properties that do not describe the content of bag buffers, which are out of order. We encode configurations $\gamma = (\mathbf{l}, b_1, \ldots, b_{|I_{\mathfrak{S}}|})$ with $[\gamma] = \mathbf{l} \cdot \# \cdot b_{\sigma(1)} \cdot \# \cdot \ldots \cdot \# \cdot b_{\sigma(|I_{\mathfrak{S}}^F|)}$, with $\sigma : |I_{\mathfrak{S}}^F| \to |I_{\mathfrak{S}}|$ being a function defined such that $\sigma(i)$ is the buffer index of the i-th FIFO buffer. This encoding is close to QDDs [5].

For a regular property P, and a system $\mathfrak{S}$, let $\mathcal{A}(\mathfrak{S})$ the automaton recognising the encoding of configurations $\mathfrak{S}$ satisfying P. We will define an automaton $\mathcal{A}_{P(\mathfrak{S})}$ recognising executions leading to configurations satisfying P (configurations that are in $P(\mathfrak{S})$). Intuitively, it works by recognising the content of each buffer independently. It does so by encoding in its states the state of $\mathcal{A}(\mathfrak{S})$ corresponding to the accepting encoding of each buffer. A 'pebble' per buffer is placed non-deterministically on a state of $\mathcal{A}(\mathfrak{S})$ and an unmatched send action $i!v$ (contributing to the content of b_i) is accepted only if there is a transition accepting v from the state marked by the i-th pebble. Let $\mathcal{A}_{P(\mathfrak{S})} = \left(L_{P(\mathfrak{S})}, \delta_{P(\mathfrak{S})}, L^0_{P(\mathfrak{S})}, F_{P(\mathfrak{S})}\right)$ be a non-deterministic finite state automaton over the alphabet $\Sigma_{\mathfrak{S}}$ of communications where the set of control states is $L_{P(\mathfrak{S})} = L_{\mathfrak{S}} \times L_{\mathfrak{S}} \times L_{\mathcal{A}}^{|I_{\mathfrak{S}}^F|} \times L_{\mathcal{A}}^{|I_{\mathfrak{S}}^F|}$, each control state $(\mathbf{l}_{\mathfrak{S}}, \mathbf{l}_f, \mathbf{l}_{\mathcal{A}}, \mathbf{l}_I)$ corresponds to a situation where:

- the current control state of $\mathfrak{S}$ is $\mathbf{l}_{\mathfrak{S}}$,
- the target control state is $\mathbf{l}_f$,
- the i-th pebble is currently on state $l_{\mathcal{A},i}$ of $\mathcal{A}(\mathfrak{S})$,
- $\mathbf{l}_I$ is a copy of the initial positions of the pebbles.

A state $(\mathbf{l}_{\mathfrak{S}}, \mathbf{l}_f, \mathbf{l}_{\mathcal{A}}, \mathbf{l}_I)$ is initial if:

- $\mathbf{l}_{\mathcal{A}} = \mathbf{l}_I$,
- $l_{\mathcal{A},1} \in \delta^*_{\mathcal{A}(\mathfrak{S})}(l_{\mathcal{A},0}, \mathbf{l}_F \cdot \#)$,
- $\mathbf{l}_{\mathfrak{S}} = \mathbf{l}^0_{\mathfrak{S}}$.

A control state is accepting if:

- $\mathbf{l}_{\mathfrak{S}} = \mathbf{l}_F$,
- for all $i \in \{1, \ldots, |I_{\mathfrak{S}}^F| - 1\}$, $\left(l_{\mathcal{A}(\mathfrak{S}),i}, \#, l_{\mathcal{A}(\mathfrak{S}),i+1}\right) \in \delta_{\mathcal{A}(\mathfrak{S})}$, and
- $l_{\mathcal{A},|I_{\mathfrak{S}}^F|} \in F_{\mathcal{A}(\mathfrak{S})}$.

Finally, a transition $\left(\left(\mathbf{l}_{\mathfrak{S}}, \mathbf{l}_f, \mathbf{l}_{\mathcal{A}}, \mathbf{l}_I\right), c, \left(\mathbf{l}'_{\mathfrak{S}}, \mathbf{l}'_f, \mathbf{l}'_{\mathcal{A}}, \mathbf{l}'_I\right)\right) \in \delta_{P(\mathfrak{S})}$ if:

- $\mathbf{l}_F = \mathbf{l}'_F$
- $\mathbf{l}_I = \mathbf{l}'_I$
- $\exists \mathbf{b}, \mathbf{b}'$ such that $(\mathbf{l}_{\mathfrak{S}}, \mathbf{b}) \overset{c}{\underset{\mathfrak{S}}{\Longrightarrow}} (\mathbf{l}'_{\mathfrak{S}}, \mathbf{b}')$,
- if $c = i!v$ and $i \in I^F_{\mathfrak{S}}$, then $\left(l_{\mathcal{A},\sigma(i)}, v, l'_{\mathcal{A},\sigma(i)}\right) \in \delta_{\mathcal{A}}$ and for all $j \in I^F_{\mathfrak{S}}, j \neq i$, $l_{\mathcal{A},\sigma(j)} = l'_{\mathcal{A},\sigma(j)}$; else, $\mathbf{l}_{\mathcal{A}} = \mathbf{l}'_{\mathcal{A}}$.

Theorem 2. *Let $\mathfrak{S}$ be an RSC system, and P a regular property, it is decidable whether $\mathfrak{S}$ is P safe.*

Proof. Let $\mathfrak{S}$ be an RSC system, $\mathcal{A}_{rsc}(\mathfrak{S})$ recognises all RSC executions of $\mathfrak{S}$, and $\mathcal{A}_{P(\mathfrak{S})}$ recognises all executions of $\mathfrak{S}$ leading to a configuration $\gamma \in P(\mathfrak{S})$; therefore $\mathcal{L}\left(\mathcal{A}_{rsc}(\mathfrak{S}) \cap \mathcal{A}_{P(\mathfrak{S})}\right) = \varnothing$ iff $\mathfrak{S}$ is P safe. □

Section 4.2 of [12] provides a non exhaustive list of regular safety properties.

B Extended benchmark results

Tables 3 and 4 show more results of membership testing. We can see that a majority the examples showcased in [19] and [3] is RSC.

Protocol	$\lvert\mathbb{P}\rvert$	S	T	RSC	t_{rsc}	k-MC	t_{kmc}
Client-Server-Logger [19]	3	11	12	No	3	Yes	17
4 players game [36]	4	13	16	Yes	13	Yes	20
Bargain [36]	3	9	8	Yes	4	Yes	35
Filter collaboration [47]	2	6	10	Yes	4	Yes	33
Alternating bit [35]	2	12	15	Yes	9	Yes	24
TPMContract v2 [34]	2	10	14	Yes	4	Yes	31
Sanitary agency [44]	4	25	30	Yes	15	Yes	39
Logistic [40]	4	26	26	Yes	8	Yes	32
Cloud system v4 [33]	4	14	16	Yes	6	Yes	22
Commit protocol [6]	4	12	12	Yes	4	Yes	15
Elevator [6]	3	13	23	No	7	Yes	41
Dev system [42]	4	22	23	Yes	7	Yes	17
Fibonacci [21]	2	6	6	Yes	3	Yes	17
SH [21]	3	22	30	Yes	18	Yes	33
Travel agency [21]	3	17	20	Yes	8	Yes	15
SMTP [16, 21]	2	64	108	Yes	17	Yes	34
HTTP [17]	2	12	48	Yes	17	Yes	28

Table 3: Comparison between the membership results of ReSCu and KMC. $\lvert\mathbb{P}\rvert$ is the number of participants, S the number of states, and T the number of transitions. t_{rsc} and t_{kmc} are the time (in ms) of execution of ReSCu and KMC respectively.

Protocol	$\|\mathbb{P}\|$	S	T	RSC	t_{rsc}	k	t_{stabc}
Estelle specification [18]	2	7	9	No	5	max	82,625
News server [41]	2	10	10	No	5	3	54,507
Client/Server [8]	2	6	10	Yes	4	1	26,130
CFSM system [18]	2	6	7	No	4	max	81,911
Promela program (1) [37]	2	6	6	No	4	2	39,905
Promela program (2) [38]	2	6	7	Yes	4	max	81,555
Web services [31]	3	13	12	Yes	5	2	53,084
Trade system [30]	3	12	12	Yes	5	1	34,726
FTP transfer [7]	3	20	17	Yes	6	4	89,465
Client/Server [28]	3	15	15	Yes	5	2	53,040
Mars explorer [24]	3	36	34	Yes	9	3	73,517
Online computer sale [29]	3	26	26	Yes	8	2	53,112
E-museum [27]	4	19	24	Yes	8	3	89,561
Client/supplier [26]	3	31	33	Yes	9	2	53,094
Restaurant service [1]	3	16	16	No	5	2	52,793
Travel agency [46]	3	34	38	Yes	10	4	102,494
Vending machine [32]	3	15	14	Yes	5	2	53,062
Travel agency [23]	3	43	56	No	13	3	71,339
Train station [45]	4	20	18	Yes	8	2	66,030
Factory job manager [25]	4	20	20	Yes	7	2	65,774
Bug report repository [13]	4	11	11	Yes	4	max	134,796
Cloud application [33]	4	8	10	No	6	max	134,655
Sanitary agency [43]	4	35	42	Yes	30	3	88,927
SQL server [22]	4	33	38	Yes	13	3	90,553
SSH [20]	4	27	28	Yes	7	2	43,855
Booking system [39]	5	45	50	Yes	48	2	78,625

Table 4: Comparison between the membership results of ReSCu and STABC, using FIFO buffers and 'strong equivalence'. $|\mathbb{P}|$ is the number of participants, S the number of states, and T the number of transitions. max means the arbitrary limit for k, set at 10, was reached. t_{rsc} and t_{stabc} are the time (in ms) of execution of ReSCu and STABC respectively.

Additional References

[23] Amel Bennaceur, Chris Chilton, Malte Isberner and Bengt Jonsson. 'Automated Mediator Synthesis: Combining Behavioural and Ontological Reasoning'. In: *Software Engineering and Formal Methods - 11th International Conference, SEFM, Proceedings*. Vol. 8137. Lecture Notes in Computer Science. Springer, 2013, pp. 274–288. DOI: 10.1007/978-3-642-40561-7_19.

[24] Antonio Brogi and Razvan Popescu. 'Automated Generation of BPEL Adapters'. In: *Service-Oriented Computing, ICSOC, 4th International Conference, Proceedings*. Vol. 4294. Lecture Notes in Computer Science. Springer, 2006, pp. 27–39. DOI: 10.1007/11948148_3.

[25] Tevfik Bultan, Chris Ferguson and Xiang Fu. 'A Tool for Choreography Analysis Using Collaboration Diagrams'. In: *IEEE International Conference on Web Services, ICWS*. IEEE Computer Society, 2009, pp. 856–863. DOI: 10.1109/ICWS.2009.100.

[26] Javier Cámara, José Antonio Martín, Gwen Salaün, Carlos Canal and Ernesto Pimentel. 'Semi-Automatic Specification of Behavioural Service Adaptation Contracts'. In: *Electron. Notes Theor. Comput. Sci.* 264.1 (2010), pp. 19–34. DOI: 10.1016/j.entcs.2010.07.003.

[27] Carlos Canal, Pascal Poizat and Gwen Salaün. 'Model-Based Adaptation of Behavioral Mismatching Components'. In: *IEEE Transactions on Software Engineering, TSE* 34.4 (2008), pp. 546–563. DOI: 10.1109/TSE.2008.31.

[28] Carlos Canal, Pascal Poizat and Gwen Salaün. 'Synchronizing Behavioural Mismatch in Software Composition'. In: *Formal Methods for Open Object-Based Distributed Systems, 8th IFIP WG 6.1 International Conference, FMOODS, Proceedings*. Vol. 4037. Lecture Notes in Computer Science. Springer, 2006, pp. 63–77. DOI: 10.1007/11768869_7.

[29] Javier Cubo, Gwen Salaün, Carlos Canal, Ernesto Pimentel and Pascal Poizat. 'A Model-Based Approach to the Verification and Adaptation of WF/.NET Components'. In: *4th International Workshop on Formal Aspects of Component Software, FACS, Proceedings*. Vol. 215. Electronic Notes in Theoretical Computer Science. Elsevier, 2007, pp. 39–55. DOI: 10.1016/j.entcs.2008.06.020.

[30] Pierre-Malo Deniélou and Nobuko Yoshida. 'Multiparty Session Types Meet Communicating Automata'. In: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS, Proceedings*. Vol. 7211. Lecture Notes in Computer Science. Springer, 2012, pp. 194–213. DOI: 10.1007/978-3-642-28869-2_10.

[31] Xiang Fu, Tevfik Bultan and Jianwen Su. 'Analysis of interacting BPEL web services'. In: *13th international conference on World Wide Web, WWW, Proceedings*. ACM, 2004, pp. 621–630. DOI: 10.1145/988672.988756.

[32] Christian Gierds, Arjan J. Mooij and Karsten Wolf. 'Reducing Adapter Synthesis to Controller Synthesis'. In: *IEEE Transasctions on Services Computing* 5.1 (2012), pp. 72–85. DOI: 10.1109/TSC.2010.57.

[33] Matthias Güdemann, Gwen Salaün and Meriem Ouederni. 'Counterexample Guided Synthesis of Monitors for Realizability Enforcement'. In: *Automated Technology for Verification and Analysis - 10th International Symposium, ATVA, Proceedings*. Vol. 7561. Lecture Notes in Computer Science. Springer, 2012, pp. 238–253. DOI: 10.1007/978-3-642-33386-6_20.

[34] Sylvain Hallé and Tevfik Bultan. 'Realizability analysis for message-based interactions using shared-state projections'. In: *18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Proceedings*. ACM, 2010, pp. 27–36. DOI: 10.1145/1882291.1882298.

[35] *Introduction to protocol egineering*. 2006.

[36] Julien Lange, Emilio Tuosto and Nobuko Yoshida. 'From Communicating Machines to Graphical Choreographies'. In: *42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, Proceedings*. ACM, 2015, pp. 221–232. DOI: 10.1145/2676726.2676964.

[37] Stefan Leue, Richard Mayr and Wei Wei. 'A Scalable Incomplete Test for Message Buffer Overflow in Promela Models'. In: *Model Checking Software, 11th International SPIN Workshop, Proceedings*. Vol. 2989. Lecture Notes in Computer Science. Springer, 2004, pp. 216–233. DOI: 10.1007/978-3-540-24732-6_16.

[38] Stefan Leue, Alin Stefanescu and Wei Wei. 'Dependency Analysis for Control Flow Cycles in Reactive Communicating Processes'. In: *Model Checking Software, 15th International SPIN Workshop, Proceedings*. Vol. 5156. Lecture Notes in Computer Science. Springer, 2008, pp. 176–195. DOI: 10.1007/978-3-540-85114-1_14.

[39] Radu Mateescu, Pascal Poizat and Gwen Salaün. 'Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques'. In: *Service-Oriented Computing, ICSOC, 6th International Conference, Proceedings*. Vol. 5364. Lecture Notes in Computer Science. 2008, pp. 84–99. DOI: 10.1007/978-3-540-89652-4_10.

[40] OMG. *Business Process Model and Notation*. 2018.

[41] Meriem Ouederni, Gwen Salaün and Tevfik Bultan. 'Compatibility Checking for Asynchronously Communicating Software'. In: *Formal Aspects of Component Software - 10th International Symposium, FACS, Revised Selected Papers*. Vol. 8348. Lecture Notes in Computer Science. Springer, 2013, pp. 310–328. DOI: 10.1007/978-3-319-07602-7_19.

[42] Roly Perera, Julien Lange and Simon J. Gay. 'Multiparty Compatibility for Concurrent Objects'. In: *Ninth workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES, Proceedings*. Vol. 211. EPTCS. 2016, pp. 73–82. DOI: 10.4204/EPTCS.211.8.

[43] Gwen Salaün, Lucas Bordeaux and Marco Schaerf. 'Describing and Reasoning on Web Services using Process Algebra'. In: *IEEE International Conference on Web Services, ICWS, Proceedings*. IEEE Computer Society, 2004, p. 43. DOI: 10.1109/ICWS.2004.1314722.

[44] Gwen Salaün, Lucas Bordeaux and Marco Schaerf. 'Describing and reasoning on Web Services using Process Algebra'. In: *International Journal of Business Process Integration and Management* 1.2 (2006), pp. 116–128. DOI: 10.1504/IJBPIM.2006.010025.

[45] Gwen Salaün, Tevfik Bultan and Nima Roohi. 'Realizability of Choreographies Using Process Algebra Encodings'. In: *IEEE Transactions on Services Computing* 5.3 (2012), pp. 290–304. DOI: 10.1109/TSC.2011.9.

[46] Ricardo Seguel, Rik Eshuis and Paul W. P. J. Grefen. 'Generating Minimal Protocol Adaptors for Loosely Coupled Services'. In: *IEEE International Conference on Web Services, ICWS*. IEEE Computer Society, 2010, pp. 417–424. DOI: 10.1109/ICWS.2010.14.

[47] Daniel M. Yellin and Robert E. Strom. 'Protocol Specifications and Component Adaptors'. In: *ACM Transactions on Programming Languages and Systems, TOPLAS* 19.2 (1997), pp. 292–333. DOI: 10.1145/244795.244801.