Towards Practical Protocol Verification via Minimal Orchestration in ACP

Bas van den Heuvel
Jorge A. Pérez
University of Groningen
Groningen, The Netherlands

Summary. We report ongoing work on the analysis of the *protocols* that pervade concurrent and distributed software. These protocols are key to ensure that communicating programs interact correctly, without communication errors and deadlocks. We focus on *multiparty session types (MPST)* [4], an approach that uses governing multiparty protocols as types to verify the correctness of message-passing programs. Rather than the π-calculus, we target the verification of programs by relying on ACP (the *Algebra of Communicating Processes*) [1] as specification language; rather than typing, we aim to adopt *model checking* using the mCRL2 toolset [2, 3].

A Motivating Example. Consider a protocol between participants Alice, Bob, and Coin. The protocol specifies a coin flipping game between Alice and Bob: Alice picks either heads or tails and Bob flips a Coin; if Alice guessed correctly, they win and try again.

In MPST, protocols are specified as *global types*. The following global type expresses our coin flipping game with participants a (Alice), b (Bob), and c (Coin):

$$G := \mu X.a \twoheadrightarrow b \left\{ \begin{array}{l} \mathsf{heads}\langle \mathsf{int} \rangle.c \twoheadrightarrow b \left\{ \begin{array}{l} \mathsf{heads}.b \twoheadrightarrow a\{\mathsf{win}.X\}, \\ \mathsf{tails}.b \twoheadrightarrow a\{\mathsf{lose}.\mathsf{end}\} \end{array} \right\}, \\ \mathsf{tails}\langle \mathsf{int} \rangle.c \twoheadrightarrow b \left\{ \begin{array}{l} \mathsf{heads}.b \twoheadrightarrow a\{\mathsf{lose}.\mathsf{end}\}, \\ \mathsf{tails}.b \twoheadrightarrow a\{\mathsf{win}.X\} \end{array} \right\} \end{array} \right\}$$

The global type G defines a recursive protocol (μX). First, a sends to b ($a \twoheadrightarrow b$) a choice between the labels "heads" and "tails" along with an integer bet ($\langle \mathsf{int} \rangle$). In both cases, c sends to b a choice between "heads" and "tails". Then, b sends to a the label "win" or the label "lose", dependending on the initial choice by a and the consecutive choice by c. In the "win" cases, the protocol repeats by means of a recursive call (X), and in the "lose" cases, the protocol ends (end).

MPST uses global types to verify the correctness of the implementation of protocol participants as distributed processes that communicate asynchronously. In this context, a participant's process implementation is correct if it satisfies all of *protocol fidelity* (the process acts as stipulated by the protocol), *communication safety* (no errors or mismatches in messages), and *deadlock freedom* (the process never gets stuck waiting for another process).

AGERE, 2021, Chicago, Illinois, USA
2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Our Proposed Approach. We aim to verify the correctness of participant implementations by means of *model checking*. Our innovation is the use of ACP as the specification language, allowing us to use mCRL2 to verify that participant implementations conform to the given global protocol.

Unlike approaches based on the π-calculus, processes in ACP are allowed to interact by a *communication function*, which defines synchronizations of parallel actions. For example, a communication function that maps "$a[\ell] \mid b(\ell) \mapsto ab\langle \ell \rangle$" says that $a[\ell]$ and $b(\ell)$ may synchronize to perform action $ab\langle \ell \rangle$. Processes can be *encapsulated* to force these synchronizations, e.g. disabling the actions $a[\ell]$ and $b(\ell)$ from being performed by themselves, forcing a synchronization as $ab\langle \ell \rangle$. In ACP, there is no concept of "output" or "input"; we refer to actions of the forms $a[\ldots]$, $b(\ldots)$, and $ab\langle \ldots \rangle$ as outputs, inputs, and communications, respectively.

One key idea is to *extract* the communication function from a given global type and use it to *orchestrate* the interactions between participant implementations. By forcing specific synchronization actions derived from the global type, we ensure that participant implementations communicate following the intended protocol. We argue that this orchestration is *minimal*, in the sense that it is the least amount of intervention necessary to ensure protocol fidelity, while avoiding synchronizations with a centralized component. Indeed, in our approach, the orchestrator is not an additional component but is subsumed by the communication function.

For this to work, we number each exchange in the global type in depth-first order. In G, this would work as follows:

$$\mu X.a \overset{1}{\twoheadrightarrow} b \left\{ \begin{array}{l} \mathsf{heads}\langle \mathsf{int} \rangle.c \overset{2}{\twoheadrightarrow} b \left\{ \begin{array}{l} \mathsf{heads}.b \overset{3}{\twoheadrightarrow} a\{\mathsf{win}.X\}, \\ \mathsf{tails}.b \overset{4}{\twoheadrightarrow} a\{\mathsf{lose}.\mathsf{end}\} \end{array} \right\}, \\ \mathsf{tails}\langle \mathsf{int} \rangle.c \overset{5}{\twoheadrightarrow} b \left\{ \begin{array}{l} \mathsf{heads}.b \overset{6}{\twoheadrightarrow} a\{\mathsf{lose}.\mathsf{end}\}, \\ \mathsf{tails}.b \overset{7}{\twoheadrightarrow} a\{\mathsf{win}.X\} \end{array} \right\} \end{array} \right\}$$

Each output and input action by a participant implementation should then be annotated with the number of the associated global exchange. In G, the action for, e.g., a's initial output of label "heads" and integer 42 should be $a_1[\mathsf{heads}, 42]$.

The following are example ACP implementations P, Q, and R of a, b, and c in G, respectively:

$$\begin{aligned} P := {} & a_1[\mathsf{heads}, 42] \,.\, (a_3(\mathsf{win}) \,.\, P + a_4(\mathsf{lose})) \\ & + a_1[\mathsf{tails}, 5] \,.\, (a_6(\mathsf{lose}) + a_7(\mathsf{win}) \,.\, P) \end{aligned}$$

$$Q \coloneqq \sum_{x\in\mathbb{Z}} b_1(\mathsf{heads}, x) \,.\, \begin{pmatrix} b_2(\mathsf{heads}) \,.\, b_3[\mathsf{win}] \,.\, Q \\ + b_2(\mathsf{tails}) \,.\, b_4[\mathsf{lose}] \end{pmatrix} + \sum_{x\in\mathbb{Z}} b_1(\mathsf{tails}, x) \,.\, \begin{pmatrix} b_5(\mathsf{heads}) \,.\, b_6[\mathsf{lose}] \\ + b_5(\mathsf{tails}) \,.\, b_7[\mathsf{win}] \,.\, Q \end{pmatrix}$$

$$R \coloneqq c_2[\mathsf{heads}] \,.\, R + c_2[\mathsf{tails}] + c_5[\mathsf{heads}] + c_5[\mathsf{tails}] \,.\, R$$

Processes consist of actions ($a_1[\ldots]$), non-deterministic choices (+) between subprocesses, and sequential compositions (.) of subprocesses. A process recursively loops by referring to its identifier, such as in P. The sum ($\sum$) in Q defines a non-deterministic choice over a (possibly infinite) set.

To fully implement G in ACP, we derive a communication function γ from G. For each protocol exchange, γ defines a synchronization between (i) a properly numbered output action by the sender and (ii) a properly numbered input action by the recipient. For example, the communication function γ derived from G includes the following synchronizations:

$$\begin{aligned}
\gamma(a_1[\mathsf{heads}, x] \mid b_1(\mathsf{heads}, x)) &= ab_1\langle \mathsf{heads}, x\rangle && \forall x \in \mathbb{Z} \\
\gamma(a_1[\mathsf{tails}, x] \mid b_1(\mathsf{tails}, x)) &= ab_1\langle \mathsf{tails}, x\rangle && \forall x \in \mathbb{Z} \\
\gamma(c_2[\mathsf{heads}] \mid b_2(\mathsf{heads})) &= cb_2\langle \mathsf{heads}\rangle \\
\gamma(c_2[\mathsf{tails}] \mid b_2(\mathsf{tails})) &= cb_2\langle \mathsf{tails}\rangle \\
\gamma(b_3[\mathsf{win}] \mid a_3(\mathsf{win})) &= ba_3\langle \mathsf{win}\rangle \\
\gamma(b_4[\mathsf{lose}] \mid a_4(\mathsf{lose})) &= ba_4\langle \mathsf{lose}\rangle
\end{aligned}$$

To force the synchronization of output and input actions associated to G, we then simply disable any output, input, and communication actions that are not in the image of γ by means of encapsulation. The resulting complete implementation S of G is then as follows:

$$S \coloneqq \partial_H(P \parallel Q \parallel R)$$

Here, $\partial_H(\ldots)$ denotes the encapsulation of the actions in the set H, and $T \parallel U$ denotes the concurrent interleaving and synchronization of actions in T and U (i.e. the *merge* of T and U). In the case of G, the set H contains, e.g., the actions $a_2[\mathsf{heads}]$, $b_3(\mathsf{lose})$, and $ba_4\langle\mathsf{win}\rangle$.

The operational semantics of ACP is defined in terms of labelled transitions. The complete implementation S of G has, e.g., the following transitions (omitting intermediate processes between transitions):

$$S \xrightarrow{ab_1\langle \mathsf{heads},42\rangle} \xrightarrow{cb_2\langle \mathsf{heads}\rangle} \xrightarrow{ba_3\langle \mathsf{win}\rangle} S$$

$$S \xrightarrow{ab_1\langle \mathsf{tails},5\rangle} \xrightarrow{cb_5\langle \mathsf{heads}\rangle} \xrightarrow{ba_6\langle \mathsf{lose}\rangle} \checkmark$$

Here, ✓ denotes succesful termination.

Note that the processes P, Q, and R are *correct* implementations for the participants in G, in the sense that they respect the causality between exchanges in G (a participant may not output before a preceding input), and that they account for each possible label when doing an input action. Let us consider what happens if we were to use alternative implementations Q' and R' of b and c in G, respectively. In this case, the implementation of b only accounts for the case in which a wins, and c is implemented as a single-sided coin.

$$Q' \coloneqq \sum_{x\in\mathbb{Z}} b_1(\mathsf{heads}, x) \,.\, b_2(\mathsf{heads}) \,.\, b_3[\mathsf{win}] \,.\, Q' + \sum_{x\in\mathbb{Z}} b_1(\mathsf{tails}, x) \,.\, b_5(\mathsf{tails}) \,.\, b_7[\mathsf{win}] \,.\, Q'$$

$$R' \coloneqq c_2[\mathsf{heads}] \,.\, R' + c_5[\mathsf{heads}]$$

$$S' \coloneqq \partial_H(P \parallel Q' \parallel R')$$

In case a chooses "heads", things proceed normally, since the coin will indeed land on "heads":

$$S' \xrightarrow{ab_1\langle \mathsf{heads},42\rangle} \xrightarrow{cb_2\langle \mathsf{heads}\rangle} \xrightarrow{ba_3\langle \mathsf{win}\rangle} S'$$

However, if a chooses "tails", b does not account for receiving "heads", so the process transitions to a deadlock state (δ) which has no further transitions:

$$S' \xrightarrow{ab_1\langle \mathsf{tails},5\rangle} \delta$$

Note that while Q' is an incorrect implementation of b in G (it does not account for all possible labels it could receive from c), R' is a correct implementation of c in G: an implementation may provide only some of the labels it may send.

Current and Future Work. We are currently developing methods of verifying the correctness of participant implementations by means of model checking. The labelled transition system of ACP is suitable for model checking with the *modal μ-calculus (Lμ)* [6]: in the mCRL2 toolset, Lμ's modalities say that given sequences of actions can or must happen, and fixpoint operators enable reasoning about infinite loops of actions. We are working on deriving Lμ formulas from global types to verify, e.g., protocol fidelity and deadlock freedom of arbitrary participant implementations. The work by Lange and Yoshida on generating Lμ formulas from session types [7] may provide us with a headstart. Scalas and Yoshida's approach of model checking MPST in mCRL2 [8] is also related, but they focus on types instead of implementations.

In the future, we would like to automate our developments by developing a toolset using, e.g., the metaprogramming language Rascal [5]. Message-passing in MPST is intended to be *asynchronous*, whereas communication in ACP is inherently *synchronous*. We are considering methods of modelling asynchronous communication in ACP, such as through output buffers. We also exploring ways of modelling *delegation*: the ability to pass channel endpoints between participants.

An important aspect of MPST is the expressivity of global types, i.e. the implementability of global types in a given approach. Interestingly, usual approaches based on typing and local projection do not support our running example. Based on this evidence, we intend to investigate theoretical results that delineate the expressivity of our approach.

Acknowledgements

We thank the reviewers for their detailed feedback, which has been very useful to prepare our talk at the workshop.

References

[1] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1):109–137, January 1984. doi:10.1016/S0019-9958(84)80025-X.

[2] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. The mCRL2 Toolset for Analysing Concurrent Systems. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 21–39, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-17465-1_2.

[3] Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, Cambridge, MA, USA, August 2014.

[4] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM*, 63(1), March 2016. doi:10.1145/2827695.

[5] Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177, September 2009. doi:10.1109/SCAM.2009.28.

[6] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, January 1983. doi:10.1016/0304-3975(82)90125-6.

[7] Julien Lange and Nobuko Yoshida. Characteristic Formulae for Session Types. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 833–850, Berlin, Heidelberg, 2016. Springer. doi:10.1007/978-3-662-49674-9_52.

[8] Alceste Scalas and Nobuko Yoshida. Less is more: Multiparty session types revisited. *Proceedings of the ACM on Programming Languages*, 3(POPL):30:1–30:29, January 2019. Revised, extended version at https://www.doc.ic.ac.uk/research/technicalreports/2018/DTRS18-6.pdf. doi:10.1145/3290343.