A theory of composing protocols

Laura Bocchi[a], Dominic Orchard[a,b], and A. Laura Voinea[c]

a University of Kent, UK

b University of Cambridge, UK

c University of Glasgow, UK

Abstract In programming, protocols are everywhere. Protocols describe the pattern of interaction (or communication) between software systems, for example, between a user-space program and the kernel or between a local application and an online service. Ensuring conformance to protocols avoids a significant class of software errors. Subsequently, there has been a lot of work on verifying code against formal protocol specifications. The pervading approaches focus on distributed settings involving parallel composition of processes within a single monolithic protocol description. However we observe that, at the level of a single thread/process, modern software must often implement a number of clearly delineated protocols at the same time which become dependent on each other, e.g., a banking API and one or more authentication protocols. Rather than plugging together modular protocol-following components, the code must re-integrate multiple protocols into a single component.

We address this concern of combining protocols via a novel notion of 'interleaving' composition for protocols described via a process algebra. User-specified, domain-specific constraints can be inserted into the individual protocols to serve as 'contact points' to guide this composition procedure, which outputs a single combined protocol that can be programmed against. Our approach allows an engineer to then program against a number of protocols that have been composed (re-integrated), reflecting the true nature of applications that must handle multiple protocols at once.

We prove various desirable properties of the composition, including behaviour preservation: that the composed protocol implements the behaviour of both component protocols. We demonstrate our approach in the practical setting of Erlang, with a tool implementing protocol composition that both generates Erlang code from a protocol and generates a protocol from Erlang code. This tool shows that, for a range of sample protocols (including real-world examples), a modest set of constraints can be inserted to produce a small number of candidate compositions to choose from.

As we increasingly build software interacting with many programs and subsystems, this new perspective gives a foundation for improving software quality via protocol conformance in a multi-protocol setting.

ACM CCS 2012

- **Software and its engineering → Specification languages**;

Keywords Distributed protocols, Composition, Engineering, Process-calculi

The Art, Science, and Engineering of Programming

Submitted June 1, 2022

Published October 15, 2022

DOI 10.22152/programming-journal.org/2023/7/6

1 Introduction

Protocols are everywhere. Whenever two entities need to communicate (perhaps via function calls, or messages sent over a channel), a protocol can be used to ensure that both parties effectively exchange information. Protocols can be seen as a *specification* of communication, and as such have been leveraged for the purposes of verification in programming languages, e.g., session types [23, 24, 7, 25], choreographies [11, 12, 37], typestate [40], behavioural types in general [28, 19], and more.

There may be many protocols that a program has to conform to, capturing different interactions between different parts of a system. Here we use the term *protocol* to denote a specification of the interaction patterns between different system components. For example, when considering distributed systems, a protocol may describe the causalities and dependencies of the communication between processes. To give a more concrete intuition, an informal specification of a protocol for an e-banking system may be as follows: *The banking server repeatedly offers a menu with three options: (1) request a banking statement, which is sent back by the server, (2) request a payment, after which the client will send payment data, or (3) terminate the session.* We elaborate on this example later, using it as a motivating example.

Much of the work on systematising the process of programming against a specification assumes a monolithic view of protocols: a protocol is often given for the entire system, explaining the communication between all parties involved. This up-front, single point of definition runs contrary to the human aspects of real-world programming, in which a programmer gradually pieces together their code, perhaps heavily leveraging libraries, to reach their intended goal; programs are gradual *compositions*.

A view that is globally defined once does not reflect the real process of software composition. In contrast, a view that defines lots of local protocols or sub-protocols places the burden of configuring their interaction on the programmer: programmers must themselves work in a situation where they have to consider many smaller protocols and work out how they want dependencies between them to be resolved. Instead, we propose that a flexible, non-monolithic notion of *protocol composition* (and possibly recomposition, when a piece of code is refactored and rewritten, or reused) is needed to support the engineering of protocol-dependent code. Ideally, such a notion should support well-founded semi-automated protocol composition and support implementation with formal guarantees.

This work lays a foundation for compositional protocol engineering based on a notion of *interleaving composition* of protocols. An interleaving composition of two protocols 'weaves' them together into a single unified protocol. This differs from sequential composition, in which one protocol follows the other or one's inputs are coupled to the other's outputs. It differs from parallel composition, which traditionally (e.g., in CCS or CSP) describes a semantic interleaving of programs; our approach calculates a single syntactic protocol specification.

We address, in general terms, the question of what a correct protocol composition is, and introduce a syntactic definition of composition that characterises finite sets of *correct* interleaving compositions, each representing a 'good way' to interleave the component protocols with respect to domain-specific user-specified constraints.

The resulting approach gives a theoretical basis for protocol (re-)engineering based on a process calculus with constraint annotations. Interleaving composition has the purpose of enhancing the awareness of what a protocol means, and facilitating reasoning about its properties. We give an algorithmic implementation of interleaving composition supporting the process of defining protocols and inspecting the generated compositions, and code generation of skeletons of processes following a given protocol (composite or not). Code generation is based on Erlang/OTP `gen_statem` behaviour [1] allowing code to be migrated in subsequent compositions and reused. Correspondence of our protocol language with Finite State Machines (FSM) via directed graphs yields straightforward links between protocols and FSM-structured code.

A related line of work defines composition as *run-time weaving*, for example applying principles of aspect-oriented programming to protocol composition [41]. Unlike [41], we *statically* derive protocol compositions that enable (human/automated) reasoning and verification of their properties. Another related line of work is *automata composition* [18, 44, 21]. Team Automata [18, 44] provide several means of composing machines via synchronization on their common actions, and give a formal framework for composition. Unlike Team Automata, we express composition constraints orthogonally to communication: instead of synchronization on common actions, we use 'asserts'/'requires' as contact points for composition, and reason about the properties of a composite protocol from the perspective of the application logic. The resulting composition relation given in this work is not characterizable as one of the synchronizations of Team Automata (discussed further in Section 6).

Unlike in aforementioned works, our protocols are mono-threaded. This is not unusual in literature, e.g., session types are essentially mono-threaded [23, 24, 7]. Also real-world protocols, such as POP2, POP3, and SMTP, are described in their RFCs as single state machines and have been modelled, without parallel composition, as session types [9, 20, 27]. Still, one could use parallel composition as a basis for defining protocol compositions (as in Team Automata), and this would yield general and syntactically concise concurrent specifications. These concurrent specifications, with all their interleavings, would be harder for a human to understand than a well-specified interleaving composition. We explore an unusual approach to composition, with the purpose of supporting a process of human understanding of what protocol composition should be. Our novel approach is also reflected in the tool. The code for the composition of two protocols is not the composition of the existing implementations (plus some adaptor code) – as one would expect. The tool generates new code via: (1) automated generation of a stubs of the new composite protocol, and (2) migration of relevant parts of the old code – besides the stub infrastructures – into the new code. This yields simple mono-thread code that are still close to the protocol's structure.

1.1 Motivating example

The banking protocol discussed earlier in this section can be formally specified as S_B in Figure 1 using a process calculus notation. S_B repeatedly (via a fixed point $\mu\mathtt{t}$) offers (denoted &) three options: option statement is followed by a send action (denoted !) of a message with the bank statement, option payment is followed by a receive action

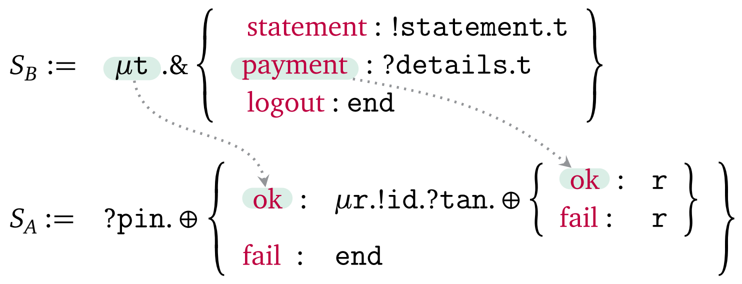
$S_B :=$ μt .& statement : !statement.t
payment : ?details.t
logout : end
$S_A :=$?pin. $\oplus$ ok : μr.!id.?tan. $\oplus$ ok : r
fail : r
fail : end

Figure 1 Banking (S_B) and PIN/TAN authentication (S_A) protocols. The arrows show the desired dependencies: entering the loop in S_B requires correct PIN authentication (i.e., at ok, first occurrence in S_A) and each payment iteration in S_B requires TAN authentication. (i.e., at ok, second occurrence in S_A).

(denoted ?) with details of the payment, and option logout is followed by termination of the protocol (denoted end). After each of the first two options, the control flow goes back to the initial state (via t).

Assume now that we want to extend S_B with two-level authentication: one level for accessing the service and one additional level for each payment transaction. Concretely, we wish to compose S_B with the PIN/TAN (Personal Identification Number/Transaction Authentication Number) protocol modelled in Figure 1 as S_A which offers two-stage authentication. The first stage is pin authentication: the server receives a pin and decides ($\oplus$) whether to continue (i.e., ok) or terminate (i.e., fail). If ok is chosen, the protocol enters a loop (i.e., μr) that manages multiple TAN authentications, supporting multiple transactions requiring an additional level of security. In the loop, the server sends an identifier id for which the client must send back a tan. The server notifies the client about the correctness of the tan with either ok or fail.

We want to compose the banking and authentication protocols into a single protocol where their actions follow a specific interleaving: access to the banking service requires a PIN authentication, and each payment instance/iteration requires an extra TAN authentication (see dotted arrows in Figure 1). This specific interleaving entails an authorization property, which we later express and ensure by using assertion annotations. Moreover, we want tools that facilitate engineering of programs implementing interleaving compositions. For example, we want to obtain a skeleton implementation for the banking and PIN/TAN protocol, and in a second stage we want to reuse the code when composing banking with a different multi-factor authentication protocol, e.g., offering other options besides TAN, such as keycard authentication.

1.2 Contributions

In Section 2, we define a process-calculus-based notation for protocols with 'assertions'. Assertions specify contact points and constraints between component protocols, to be checked statically. In Section 3, we give a definition of interleaving composition that is relational, as there may be many valid interleaved protocols (or even none). In Section 3.1.1 we provide two less restrictive definitions of interleaving composition via two additional rules, *weak branching* and *correlating branching* that capture more scenarios but enjoy a weaker fairness properties. In Section 4, we prove that our composition relation returns *correct* interleaving compositions, namely: (*behaviour preservation*) interleaving compositions only perform sequences of actions that may be performed by either of the component protocols; (*fairness*) interleaving compositions eventually execute the next available action of each protocol; (*well-assertedness*) interleaving compositions always satisfy requirements prescribed by the assertions in the protocols being composed. Thus, we establish that the composition relation produces sets of protocol compositions that are correct-by-construction. Our definition is sound but not complete, as discussed in Section 4.4. In Section 5, we introduce a tool for protocol engineering in Erlang, which implements interleaving composition, generation and protocol extraction to/from Erlang gen_statem code. Section 6 discusses related work.

2 Asserted Protocols

We introduce a language of protocol specifications to abstractly capture essential features of sequential computation: sequencing, choice, and looping. Our protocol language somewhat resembles Milner's CCS [36] or the π-calculus [39], but without parallel composition or name restriction, and has some relation to Kleene algebras [32] but we provide more general patterns of recursion via recursive binders rather than a single closure operator. Generally, two protocols can be composed in several ways, each reflecting a possible interleaving of the actions of the two protocols. Not all such interleavings are meaningful depending on the scenario or domain. The protocol language therefore includes a notion of 'assertions' which can be used to capture the behavioural constraints of a protocol to guide interleaving composition in a meaningful way; they act as a specification of minimal 'contact points' between protocols akin to pre- and post-conditions. Following an explanation of the syntax and various examples, we give an operational model to the protocol language which serves to explain both the program semantics which it abstracts, and the meaning of the assertion actions.

Definition 1 (Asserted protocols) *Asserted protocols, or just* protocols *for short, are ranged over by S and are defined as the following syntax rules:*

$$
\begin{array}{rcll}
S & ::= & p.S & \textit{action prefix} \\
& \mid & +\{\mathrm{l_i} : S_i\}_{i \in I} & \textit{branching} \\
& \mid & \mu\mathtt{t}.S & \textit{fixed-point} \\
& \mid & \mathtt{t} & \textit{recursive variable} \\
& \mid & \mathsf{end} & \textit{end} \\
& \mid & \mathsf{assert}(n).S & \textit{assert (produce)} \\
& \mid & \mathsf{require}(n).S & \textit{require} \\
& \mid & \mathsf{consume}(n).S & \textit{consume}
\end{array}
\left.\begin{array}{l} \\ \\ \\ \end{array}\right\} \text{assertion fragment}
$$

where $p \in \mathcal{P}$ ranges over prefixing actions, $\mathrm{l} \in \mathcal{L}$ ranges over labels used to label each branch of the n-ary branching construct, $\mathtt{t}$ ranges over protocol variables for recursive *protocol definitions, and $n \in \mathscr{N}$ ranges over names of logical atoms used by assertions. The sets of actions $\mathscr{P}$, labels $\mathscr{L}$, and names $\mathscr{N}$ are parameters to the language and thus can be freely chosen. Furthermore $+$ ranges over a set of operators $\mathscr{O}$ used to represent branching choice and thus can also be instantiated.*

The prefixing action provides sequential composition (in the style of process calculi). Branching is n-ary, taking the form of a set of protocol choices with a label l_i for each choice. Looping behaviour is captured via the recursive protocol variable binding $\mu\mathtt{t}$, which respects the usual rules of binders, and recursion variables $\mathtt{t}$. Protocols can be annotated with assertions to introduce guarantees $\mathsf{assert}(n)$, requirements $\mathsf{require}(n)$, and linear requirements $\mathsf{consume}(n)$: $\mathsf{assert}(n)$ introduces a true logical atom n into the scope of the following protocol, $\mathsf{require}(n)$ allows the protocol to proceed only if n is in the scope (basically $\mathsf{consume}(n)$ presupposes $\mathsf{require}(n)$), and $\mathsf{consume}(n)$ removes the truth of logical atom n from the scope of the following protocol.

We assume variables to be guarded in the standard way (they only occur under actions or branching). To simplify the theory, we assume that: (1) nested recursions are guarded, ruling out protocols of the form $\mu\mathtt{t}.\mu\mathtt{t}'.S$, with no loss of generality since $\mu\mathtt{t}.\mu\mathtt{t}'.S$ is behaviourally equivalent to $\mu\mathtt{t}.S[\mathtt{t}/\mathtt{t}']$, and (2) in $\mu\mathtt{t}.S$ variable $\mathtt{t}$ occurs free at least once in S, with no loss of generality since e.g., $\mu\mathtt{t}.?\mathtt{pay}.\mathtt{end}$ is behaviourally equivalent to $?\mathtt{pay}.\mathtt{end}$. Unless otherwise stated, we consider protocols to be closed with respect to these recursion variables.

Remark 1 (Language instantiation) *In the examples we often instantiate the prefixing actions $\mathscr{P}$ to sends* $!\mathrm{T}$ *and receives* $?\mathrm{T}$ *capturing interaction with some other concurrent program, i.e., $p \in \{!\mathrm{T}, ?\mathrm{T}\}$ where T is a type (e.g., integers, strings), and instantiate choice $+$ to a pair of* polarised choice *operators: $+ \in \{\oplus, \&\}$, either offering of a choice $\oplus$ or selecting from amongst some choices $\&$. This yields a session types-like syntax similar to the one used by Dardha, Giachino and Sangiorgi. [16].*

Examples often colour assertions green and labels purple for readability.

2.1 Assertion examples

Consider a payment process $?\mathtt{pay}.\mathtt{end}$ that receives a payment and terminates, and a dispatch process $!\mathtt{item}.\mathtt{end}$ that sends a product link and terminates. We can interleave these two protocols in two ways: $?\mathtt{pay}.!\mathtt{item}.\mathtt{end}$ (payment first) or $!\mathtt{item}.?\mathtt{pay}.\mathtt{end}$ (dispatch first). By using assertions, we can require that payment happens before dispatch: below, I_1 asserts the logical atom *paid* as a post-condition to receiving payment while in I_2 the sending action depends on the logical atom *paid* as a pre-condition, and in doing so consumes it.

$$I_1 = ?\mathtt{pay}.\mathsf{assert}(\mathit{paid}).\mathtt{end} \qquad I_2 = \mathsf{consume}(\mathit{paid}).!\mathtt{item}.\mathtt{end}$$

The only interleaving composition of I_1 and I_2 that satisfies the constraints posed by the assertions is: $?\mathtt{pay}.\mathsf{assert}(\mathit{paid}).\mathsf{consume}(\mathit{paid}).!\mathtt{item}.\mathtt{end}$.

Linear constraint $\mathsf{consume}(n)$ models a guarantee that can be used once, whereas non-linear constraint $\mathsf{require}(n)$ does not consume n. Using a mix of linear and non-linear constraints, we can model a prepaid buffet scenario where a payment remains valid (hungry) for several iterations until the meal ends (end):

$$\mu\mathtt{t}.\&\{\mathsf{hungry} : \mathsf{require}(\mathit{paid}).!\mathtt{food}.\mathtt{t},\ \mathsf{end} : \mathsf{consume}(\mathit{paid}).\mathtt{end}\}$$

Example 1 (Asserted banking and PIN/TAN) *The informal requirement on the banking and PIN/TAN example discussed in the introduction can be modelled using assertions. An asserted version of the banking protocol, given below as S'_B, uses $\mathsf{require}(\mathit{pin})$ to ensure a successful PIN authentication before accessing the banking menu; $\mathsf{consume}(\mathit{tan})$ to require one successful TAN authentication for each iteration involving a payment; and $\mathsf{consume}(\mathit{pin})$ to remove the PIN guarantee when logging out. Assertions $\mathsf{assert}(\mathit{pay})$ and $\mathsf{consume}(\mathit{pay})$ ensure TAN authentication only happens in case of payment.*

$$S'_B = \mathsf{require}(\mathit{pin}).\mu\mathtt{t}.\&\left\{\begin{array}{ll} \mathsf{statement}: & !\mathtt{statement}.\mathtt{t} \\ \mathsf{payment}: & \mathsf{assert}(\mathit{pay}).\mathsf{consume}(\mathit{tan}).?\mathtt{details}.\mathtt{t} \\ \mathsf{logout}: & \mathsf{consume}(\mathit{pin}).\mathtt{end} \end{array}\right\}$$

In the asserted authentication protocol S'_A below, $\mathsf{assert}(\mathit{pin})$ and $\mathsf{assert}(\mathit{tan})$ provide guarantees of successful PIN and TAN authentication, respectively:

$$S'_A = \ ?\mathtt{pin}.\oplus\left\{\begin{array}{ll} \mathsf{ok}: & \mathsf{assert}(\mathit{pin}).\mu\mathtt{r}.\mathsf{consume}(\mathit{pay}).!\mathtt{id}.?\mathtt{tan}.\oplus\left\{\begin{array}{ll} \mathsf{ok}: & \mathsf{assert}(\mathit{tan}).\mathtt{r} \\ \mathsf{fail}: & \mathtt{r} \end{array}\right\} \\ \mathsf{fail}: & \mathtt{end} \end{array}\right\}$$

2.2 Protocol semantics

The semantics of a protocol is given in Definition 2 in terms of an environment that keeps track of guarantees, and lets protocols progress only if stated guarantees can be met by the environment. The semantics is up to the structural equivalence rules given below, where $S[\mu\mathtt{t}.S/\mathtt{t}]$ is the one-time unfolding of $\mu\mathtt{t}.S$.

$$\mu\mathtt{t}.S \equiv S\ (\textit{where}\ \mathtt{t} \notin \mathsf{fv}(S)) \qquad \mu\mathtt{t}.S \equiv S[\mu\mathtt{t}.S/\mathtt{t}]$$

Definition 2 (Operational semantics) *The semantics of protocols is defined by a labelled transition system (LTS) over configurations of the form (A, S) where A ranges over environments $A \subseteq \mathcal{N}$ (sets of logical atoms), with transition labels $\ell ::= p \mid +\mathrm{l} \mid \mathsf{assert}(n) \mid \mathsf{require}(n) \mid \mathsf{consume}(n)$ and the transition rules below:*

$$\begin{array}{lll}
(A, p.S) \xrightarrow{p} (A, S) & & \langle\mathtt{Inter}\rangle \\
(A, +\{\mathrm{l_i} : S_i\}_{i\in I}) \xrightarrow{+\mathrm{l_j}} (A, S_j) & (j \in I) & \langle\mathtt{Branch}\rangle \\
(A, \mathsf{assert}(n).S) \xrightarrow{\mathsf{assert}(n)} (A \cup \{n\}, S) & & \langle\mathtt{Assert}\rangle \\
(A, \mathsf{require}(n).S) \xrightarrow{\mathsf{require}(n)} (A, S) & (n \in A) & \langle\mathtt{Require}\rangle \\
(A, \mathsf{consume}(n).S) \xrightarrow{\mathsf{consume}(n)} (A \setminus \{n\}, S) & (n \in A) & \langle\mathtt{Consume}\rangle \\
\dfrac{(A, S) \xrightarrow{\ell} (A', S')}{(A, \mu\mathtt{t}.S) \xrightarrow{\ell} (A', S'[\mu\mathtt{t}.S/\mathtt{t}])} & & \langle\mathtt{Rec}\rangle
\end{array}$$

Rules ⟨`Inter`⟩ and ⟨`Branch`⟩ always allow a protocol to proceed with some action, resulting in the appropriate continuation, without any effect to the environment. Rule ⟨`Assert`⟩ adds atom n to the environment. Rules ⟨`Require`⟩ and ⟨`Consume`⟩ both require the presence of atom n in the environment for the protocol to continue. Although ⟨`Require`⟩ leaves the environment unchanged, ⟨`Consume`⟩ consumes the atom n from the environment. In ⟨`Rec`⟩, $S'[\mu \mathtt{t}.S/\mathtt{t}]$ means that the recursive protocol is unfolded by substituting $\mu\mathtt{t}.S$ for $\mathtt{t}$ in S'.

We write: $(A,S) \not\rightarrow$ if $(A,S) \xrightarrow{\ell} (A',S')$ for no ℓ, A', S'; $(A,S) \xrightarrow{\vec{\ell}} (A',S')$ for a vector $\vec{\ell} = \ell_1,\dots,\ell_n$ if $(A,S) \xrightarrow{\ell_1} \dots \xrightarrow{\ell_n} (A',S')$. We say that (A',S') is *reachable* from (A,S) if $(A,S) = (A',S')$ or $(A,S) \xrightarrow{\vec{\ell}} (A',S')$ for a vector $\vec{\ell}$. We omit labels and target states where immaterial.

Definition 3 (Stuck state & progress) *State (A,S) is* stuck *if $S \not\equiv$ end and $(A,S) \not\rightarrow$. A protocol S enjoys* progress *if every state (A',S') reachable from $(\emptyset,S)$ is not stuck.*

A protocol may reach a stuck state when it does not have sufficient pre-conditions in its environment A. In Example 1, S'_B does not enjoy progress because the pre-condition expressed by require(pin) cannot be met; similarly, S'_A does not enjoy progress because of unmet pre-condition consume(pay).

2.3 Well-assertedness

Assertions are key to generating meaningful compositions of protocols. Following the labelled transitions semantics, we define a judgement which captures the pre- and post-conditions of a protocol implied by its assertions. We use the notation $A\ \{S\} A'$ reminiscent of a Hoare triple where A and A' are pre- and post-conditions of S.

Definition 4 (Well-assertedness) *Let A be a set of names.* Well-assertedness *of a protocol S with respect to A is defined below, as an inference system on judgements of the form $A\ \{S\}A'$, where A' is the set of names (logical atoms) resulting after the execution of S given the set of names A.*

$$\frac{A\ \{S\}A'}{A\ \{p.S\}A'}\,[\text{act}] \qquad \frac{\forall i \in I.\ A\ \{S_i\}A_i}{A\ \{+\{\mathrm{l}_i : S_i\}_{i\in I}\}\ \bigcap_{i\in I} A_i}\,[\text{bra}] \qquad \frac{A \cup \{n\}\ \{S\}A'}{A\ \{\mathsf{assert}(n).S\}A'}\,[\text{assert}]$$

$$\frac{A \cup \{n\}\ \{S\}A'}{A \cup \{n\}\ \{\mathsf{require}(n).S\}A'}\,[\text{require}] \qquad \frac{A \setminus \{n\}\ \{S\}A' \qquad n \in A}{A\ \{\mathsf{consume}(n).S\}A'}\,[\text{consume}]$$

$$\frac{A\ \{S\}A \cup A'}{A\ \{\mu\mathtt{t}.S\}A \cup A'}\,[\text{rec}] \qquad \frac{-}{A\ \{\mathtt{end}\}A}\,[\text{end}] \qquad \frac{-}{A\ \{\mathtt{t}\}A}\,[\text{call}]$$

We write $A\ \{S\}$ when $A\ \{S\}A'$ for some A' (i.e., when the post-condition is not of interest). We say that S is very-well-asserted *if $\emptyset\ \{S\}$. We say that a state (A,S) is well-asserted if S is well-asserted with respect to A.*

Protocols S'_A and S'_B in Example 1 are not very-well-asserted but they are well-asserted with respect to $\{pin, tan\}$ and $\{pay\}$, respectively.

We now consider some properties of well-asserted protocols. Proofs are in **??**. Firstly, protocols that do not contain assertions are very-well-asserted:

Proposition 1 (Very-well-assertedness) *If S is generated by the grammar in Definition 1 without the assertion fragment then it is very-well-asserted.*

Next, well-asserted protocols can have their environment weakened, akin to precondition weakening in Hoare logic:

Proposition 2 (Environment weakening) *If $A\ \{S\}$ and $A \subseteq A'$ then $A'\ \{S\}$. Hence, $\emptyset\ \{S\}$ implies $A\ \{S\}$ for all A.*

Next, Lemma 1 states that the redux of a well-asserted state is well-asserted, moreover the postconditions are not weakened by reduction:

Lemma 1 (Reduction preserves well-assertedness) *If $A\ \{S\} A'$ and there is a reduction $(A, S) \xrightarrow{\ell} (A'', S')$ then $\exists A''' \supseteq A'. A''\ \{S'\} A'''$.*

Lemma 2 (Well-asserted protocols are not stuck) *If $A\ \{S\}$ and S is closed with respect to recursion variables* $(\mathsf{fv}(S) = \emptyset)$ *then (A, S) is not stuck.*

Next, Lemma 3 shows that if a protocol "gets stuck", this is because it does not have enough preconditions to proceed. Thus, the protocol needs assumptions that may be provided by other protocols it could be composed with. Lemma 3 follows by induction on the length of a protocol's execution, combined with Lemmas 1 and 2.

Lemma 3 (Progress of very-well-asserted protocols) *If S is very-well-asserted (i.e., $\emptyset\ \{S\}$) and closed then it exhibits* progress.

We next introduce protocol composition, which produces protocols that are meaningful with respect to their assertions (i.e., that exhibit progress).

3 Interleaving Compositions

We compose protocols by computing syntactic interleavings. We derive the 'interleaving composition' (IC) of two protocols S_1 and S_2 via a relation with judgements of the form: $T_L; T_R; A \vdash S_1 \circ S_2 \triangleright S$ where S is the resulting composed protocol, and A is the set of names (i.e., assertions) provided by the environment to S. We let T range over recursion environments, defined as possibly empty lists of *distinct* protocol variables $\mathtt{t}$. Lists are concatenated via the , (comma) operator, which is overloaded to extend a list with a single element, e.g., written $T, \mathtt{t}$. In the judgements, we use two recursion environments T_L and T_R to keep track of the free protocol variables in S_1 and S_2 respectively in order to handle composition of recursive protocols. We use an underlining annotation $\underline{\mathtt{t}}$ to denote variables that were used to merge two recursive protocols into one recursive IC, and predicate $\mathtt{unused}(T)$ that is true if all variables in T are not used (i.e., not underlined), and false otherwise. The 'used' annotation $\underline{\mathtt{t}}$ is instrumental in handling composition of nested recursions, as explained later.

Definition 5 (Interleaving composition) *IC is defined by the judgements in Figure 2.*

$$\frac{T_L; T_R; A \vdash S_1 \circ S_2 \triangleright S}{T_L; T_R; A \vdash p.S_1 \circ S_2 \triangleright p.S} \qquad \frac{T_R; T_L; A \vdash S_2 \circ S_1 \triangleright S}{T_L; T_R; A \vdash S_1 \circ S_2 \triangleright S} \qquad [\text{act/sym}]$$

$$\frac{T_L; T_R; A \cup \{n\} \vdash S_1 \circ S_2 \triangleright S}{T_L; T_R; A \cup \{n\} \vdash \mathsf{require}(n).S_1 \circ S_2 \triangleright \mathsf{require}(n).S} \qquad [\text{require}]$$

$$\frac{T_L; T_R; A \setminus \{n\} \vdash S_1 \circ S_2 \triangleright S \qquad n \in A}{T_L; T_R; A \vdash \mathsf{consume}(n).S_1 \circ S_2 \triangleright \mathsf{consume}(n).S} \qquad [\text{consume}]$$

$$\frac{T_L; T_R; A \cup \{n\} \vdash S_1 \circ S_2 \triangleright S}{T_L; T_R; A \vdash \mathsf{assert}(n).S_1 \circ S_2 \triangleright \mathsf{assert}(n).S} \qquad [\text{assert}]$$

$$\frac{\forall i \in I \quad T_L; T_R; A \vdash S_i \circ S_2 \triangleright S'_i}{T_L; T_R; A \vdash +\{\mathrm{l}_i : S_i\}_{i\in I} \circ S_2 \triangleright +\{\mathrm{l}_i : S'_i\}_{i\in I}} \qquad [\text{bra}]$$

$$\frac{T_L, \mathtt{t}_1; T_R; A \vdash S_1 \circ \mu\mathtt{t}_2.S_2 \triangleright S \quad A\ \{\mu\mathtt{t}_1.S\}}{T_L; T_R; A \vdash \mu\mathtt{t}_1.S_1 \circ \mu\mathtt{t}_2.S_2 \triangleright \mu\mathtt{t}_1.S} \qquad \frac{A\ \{\mu\mathtt{t}.S\} \quad \mathsf{fv}(\mu\mathtt{t}.S) = \emptyset}{T_L; T_R; A \vdash \mu\mathtt{t}.S \circ \mathtt{end} \triangleright \mu\mathtt{t}.S} \qquad [\text{rec1/rec3}]$$

$$\frac{T_L; T_1, \underline{\mathtt{t}}, T_2; A \vdash S_1[\mathtt{t}/\mathtt{t}_1] \circ S_2 \triangleright S \qquad \mathtt{unused}(T_2)}{T_L; T_1, \mathtt{t}, T_2; A \vdash \mu\mathtt{t}_1.S_1 \circ S_2 \triangleright S} \qquad [\text{rec2}]$$

$$\frac{\underline{\mathtt{t}} \in T_L \vee \underline{\mathtt{t}} \in T_R}{T_L; T_R; A \vdash \mathtt{t} \circ \mathtt{t} \triangleright \mathtt{t}} \qquad \frac{-}{T_L; T_R; A \vdash \mathtt{end} \circ \mathtt{end} \triangleright \mathtt{end}} \qquad [\text{call/end}]$$

Figure 2 Rules for iterleaving composition of protocols

In Figure 2, rule [act] is for prefixes, [sym] is the commutativity rule, and [end] handles a terminated protocol. By combining [act] and [sym] one can obtain all interleavings of two sequences of actions.

Rule [require] includes the continuation of a protocol only if a required assertion n is provided by the environment. Rule [consume] is similar except the assertion is removed in the precondition's environment. Conversely, [assert] adds assertion n to the environment of the precondition. Rules [require], [assume], and [consume] may enforce a particular order in actions of an interleaving. For example, the reader can verify that the composition of ?pay.assert(p).end and consume(p).!item.end produces (only) one interleaving ?pay.assert(p).consume(p).!item.end that is obtained by applying [act], [assert], [sym], [consume], [act], and [end].

Rule [bra] is similar to [act] but the continuations are composed with each branch. For example the composition $+\{\mathrm{l}_1 : \mathtt{end},\ \mathrm{l}_2 : \mathtt{end}\} \circ\ !\mathtt{Int.end}$ with initially empty environment produces the following two interleavings:

$+\{\mathrm{l}_1 :!\mathtt{Int.end},\ \mathrm{l}_2 :!\mathtt{Int.end}\}$ (applying [bra], [sym], [act], [end])
$!\mathtt{Int.} + \{\mathrm{l}_1 : \mathtt{end},\ \mathrm{l}_2 : \mathtt{end}\}$ (applying [sym], [act], [act], [sym], [bra], [end])

Rules [rec1] and [rec2] allow two recursive protocols to be composed. The composition of two recursive protocols, say $\mu\mathtt{t}_1.S_1$ and $\mu\mathtt{t}_2.S_2$, yields a recursive protocol where the recursion body is the composition of the two recursion bodies, and only one of the two protocol variables is used, either $\mathtt{t}_1$ or $\mathtt{t}_2$. For example, the composition of $\mu\mathtt{t}_1.!\mathtt{p}_1.\mathtt{t}_1$ and $\mu\mathtt{t}_2.!\mathtt{p}_2.\mathtt{t}_2$ yields e.g.,

$\mu\mathtt{t}_1.!\mathtt{p}_1.!\mathtt{p}_2.\mathtt{t}_1$ (applying [rec1], [act], [sym], [rec2], [act], [call])
$\mu\mathtt{t}_2.!\mathtt{p}_2.!\mathtt{p}_1.\mathtt{t}_2$ (applying [sym], and proceeding as above)

Rule [rec1] picks $\mathtt{t}_1$ as name for the interleaving composition, records $\mathtt{t}_1$ as the end of the T_L list and continues with the composition of the recursion body S_1 with $\mu\mathtt{t}_2.S_2$. The premise A $\{\mu\mathtt{t}_1.S\}$ ensures well-assertedness of the *arbitrary repetition* of S, that is $\mu\mathtt{t}_1.S$ (the composition rules only check that S is well-asserted). Rule [rec2] completes the merge of two recursions, with calls to $\mathtt{t}_2$ in this instance being redirected to $\mathtt{t}_1$ (via a substitution). Variable $\mathtt{t}_1$ is in the right recursion environment $T_1, \mathtt{t}_1, T_2$, namely a list of protocol variables, followed by unused $\mathtt{t}_1$, followed by a list of unused protocol variables T_2, yielding a protocol with just one recursion. In the premise of [rec2], $\mathtt{t}_1$ in this instance becomes used.

In [rec2], condition $\mathtt{unused}(T_2)$ prevents erroneous 'flattening' of nested recursions. For instance, in the composition of $S_1 = \mu\mathtt{t}.p.\mathtt{t}$ and $S_2 = \mu\mathtt{t}_1.q.\mu\mathtt{t}_2.+\{\mathrm{l}_1 : \mathtt{t}_1, \mathrm{l}_2 : \mathtt{t}_2\}$, merging $\mathtt{t}$ with both $\mathtt{t}_1$ and $\mathtt{t}_2$ would yield the undesirable derivation $S = \mu\mathtt{t}.p.q.+\{\mathrm{l}_1 : \mathtt{t}, \mathrm{l}_2.\mathtt{t}\}$ where S does not preserve the behaviour of S_2. Behaviour preservation is formally defined later on; for now, observe that S_2 permits successive choices of the label l_2 without any intervening actions, whereas S requires an intervening q action (and p action) between any successive choices of label l_2. See **??** in **??** for some derivations of interleaving compositions of S_1 and S_2. The requirement that $\mathtt{t}$ precedes only unused variables T_2 (captured by predicate $\mathtt{unused}(T_2)$) also prevents 'criss-cross' substitutions when composing two protocols with nested recursions which can also violate behaviour preservation in similar ways to the case observed above.

Consider now the composition of a recursive protocol with a non-recursive one e.g., $S_1 = \mu\mathtt{t}.p_1.\mathtt{t}$ with $S_2 = p_2.\mathtt{end}$. We do *not* want to derive the following protocol: $S = \mu\mathtt{t}.p_1.p_2.\mathtt{t}$ The problem with S is that it allows execution $p_1, p_2, p_1, p_2, \ldots$ where action p_2 is repeatedly executed, while S_2 only prescribes one instance of p_2. Such a derivation would not preserve the behaviour of S_2. Our rules do not allow derivation of S above because rule [call] checks that the component protocols share protocol variable $\mathtt{t}$ (i.e., they are both recursive and correctly merged).

Another undesirable composition of $S_1 = \mu\mathtt{t}.p_1.\mathtt{t}$ and $S_2 = p_2.\mathtt{end}$ is one where S_1 'comes first' yielding $S' = \mu\mathtt{t}.p_1.\mathtt{t}$ which, morally, behaves as S_2 after an infinite loop. If this were a composition, it would violate a second property we discuss formally later, fairness, requiring each component protocol to be able to proceed until it terminates. S' is not derivable thanks to [rec3], which only allows a recursive protocol to be introduced in an interleaving composition when the non-recursive component has already been all merged (i.e., it is $\mathtt{end}$). We can, e.g., derive the following composition of S_1 and S_2, where the terminating protocol S_2 comes first (hence satisfying fairness):

$p_2.\mu\mathtt{t}.p_1.\mathtt{t}$ (applying [act], [sym], [rec3]).

The premise $\mathsf{fv}(\mu\mathtt{t}.S) = \emptyset$ of [rec3] prevents it being used inappropriately in case of nested recursion, e.g., to prevent composition of $\mu\mathtt{t}_1.p_1.\mu\mathtt{t}_2.p_2.\mathtt{t}_1$ and $q.\mathtt{end}$ to produce (via [rec1], [act], [sym], [rec3]) $\mu\mathtt{t}_1.p_1.q.\mu\mathtt{t}_2.p_2.\mathtt{t}_1$, which violates behaviour preservation (discussed later) by repeating an action q from a non-recursive context.

3.1 Variations on the branching rule

The branching rule of interleaving composition can be viewed as a *distributivity* property: sequential composition after a control-flow branch can be distributed inside the branches. Algebraically, we can informally describe this distributivity as follows, for a 2-way branch (*sans* labelling): $(S_1 + S_2) \circ T \equiv (S_1 \circ T) + (S_2 \circ T)$. Such a property is familiar in Kleene algebra models of programs and program reasoning [32] and monotone dataflow frameworks in static analysis [30]. Since interleaving composition generates a set of possible protocols it would be more accurate to express this property in terms of set membership rather than equality (for simplicity of the analogy, this elides the fact that each composition $\circ$ is itself a set):

$$(S_1 + S_2) \circ T \ni (S_1 \circ T) + (S_2 \circ T) \qquad \text{(distributivity)}$$

In this section we consider two variants of this distributive behaviour for composition called (1) 'weak branching' and (2) 'interchange branching' which can be summarised via the algebraic analogy as variants of distributivity, respectively:

$$(S_1 + S_2) \circ T \ni (S_1 \circ T) + S_2 \quad \wedge \quad (S_1 + S_2) \circ T \ni S_1 + (S_2 \circ T) \qquad \text{(weak)}$$
$$(S_1 + S_2) \circ (T_1 + T_2) \ni (S_1 \circ T_1) + (S_2 \circ T_2) \qquad \text{(interchange)}$$

In (weak), composition distributes inside one branch but not the other. In (interchange), composing branches with branches has a 'merging' effect on the branches rather than distributing within. (The 'interchange' terminology comes from similar properties in category theory [31]).

We motivate and discuss each variation from the protocol perspective. In the rest of this section we introduce two additional composition rules: [wbra] for weak branching, and [cbra] for interchange branching (which we will refer to as *correlating branching* as it better reflects the effects of the rule on the protocols). Note that these two variations grow the set of possible interleavings, rather than shrinking it: they provide more general composition behaviours but do not exclude the more specialised behaviours. For generality of the theory, the derivation of interleaving composition can apply any branching ([bra], [wbra], [cbra]). For practicality, our tool allows engineers to choose the kind of branching to use in any specific scenario (as shown in Section 5).

3.1.1 Weak branching for "asymmetric" guarantees

Weak branching allows partial execution of some protocols being composed even if there are not sufficient assertions to continue, as long as all protocols are completely executed in some execution path. For example, protocol S_B below needs assertion n to proceed. Assume we want to compose S_B with a protocol S_A, which can provide n in only one of its branches ok. Protocol S_A may be an authentication server, granting or blocking access to S_B depending on a password pwd. That is, for some S':

$$S_A \;::=\; ?\texttt{pwd}.\oplus\{\mathsf{ok} : \mathsf{assert}(n).\,\mathtt{end},\ \mathsf{ko} :\ \mathtt{end}\} \qquad S_B \;::=\; \mathsf{require}(n).S'$$

Since we want the actions of S_B not to be executed after selection of label ko, we want interleaving composition to generate the following protocol:

$$S_{AB} = ?\texttt{pwd}.\oplus\{\mathsf{ok} :\ \mathsf{assert}(n).\mathsf{require}(n).S',\ \mathsf{ko} :\ \mathtt{end}\}$$

Protocol S_{AB} is not attainable using the rules of Definition 5: the derivation blocks composing $\mathsf{require}(n).S'$ with the second branch's $\mathtt{end}$ in the empty environment.[1] Instead, we introduce a 'weak branching' composition rule to allow asymmetric guarantees:

Definition 6 (Weak branching) Weak branching composition *of protocols is derived using the judgements in Definition 5 and the* additional *rule [wbra]:*

$$\frac{\begin{array}{c} I = I_A \cup I_B \qquad I_A \cap I_B = \emptyset \qquad I_A \neq \emptyset \\ \forall i \in I_A.\ T_L; T_R; A \vdash S_i \circ S \triangleright S'_i \qquad \forall i \in I_B.\ T_L; T_R; A \vdash S_i \circ S \not\triangleright \ \wedge\ A\,\{S_i\} \end{array}}{T_L; T_R; A \vdash +\{\mathsf{l}_i : S_i\}_{i \in I} \circ S \triangleright +\{\mathsf{l}_i : S'_i\}_{i \in I_A} \cup \{\mathsf{l}_i : S_i\}_{i \in I_B}}$$

Precondition $I_A \neq \emptyset$ ensures that each protocol's actions are executed in at least one execution path, and is key to the fairness property introduced in Definition 9. Hereafter we denote with $\triangleright_s$ derivations obtained using the judgements in Definition 5 only and $\triangleright_w$ for derivations with the additional rule [wbra].

Example 2 (Weak IC of banking and PIN/TAN) *Consider the banking and PIN/TAN protocols in Example 1 (p. 7). Interleaving composition of S'_A and S'_B using $\triangleright_s$ returns an empty set. When using $\triangleright_w$ instead, we can derive the following interleaving composition modelling a banking/authentication protocol that satisfies the requirements specified in Section 1.1.*

$$S_{BA} = ?\mathtt{pin}.\oplus \left\{ \begin{array}{l} \mathsf{ok}:\ \mathsf{assert}(pin).\mathsf{require}(pin).\mu\mathtt{r}.\& \left\{ \begin{array}{l} \mathsf{payment}:\ S_{TAN}, \\ \mathsf{statement}: !\mathtt{statement}.\mathtt{r}, \\ \mathsf{logout}:\ \mathsf{consume}(pin).\ \mathtt{end} \end{array} \right\} \\ \\ \mathsf{fail}:\ \mathtt{end} \end{array} \right\}$$

$$S_{TAN} = \mathsf{assert}(pay).\mathsf{consume}(pay).!\mathtt{id}.?\mathtt{tan}.\oplus \left\{ \begin{array}{l} \mathsf{ok}: \mathsf{assert}(tan).\ \mathsf{consume}(tan). \\ \qquad ?\mathtt{details}.\mathtt{r}, \\ \mathsf{fail}: \mathtt{r} \end{array} \right\}$$

3.1.2 Correlating branching

Correlating branching allows two protocols to be composed by 'correlating' each branch of one with at least one branch of the other.

Consider two branching protocols: S_1 offering two services $\mathsf{s1}$ and $\mathsf{s2}$, and S_2 offering two kinds of payment $\mathsf{p1}$ and $\mathsf{p2}$. When composing S_1 and S_2, we can correlate $\mathsf{s1}$ with $\mathsf{p1}$, and $\mathsf{s2}$ with $\mathsf{p2}$, using assertions:

$$\begin{array}{rcl} S_1 & = & \oplus\{\mathsf{s1} : \mathsf{assert}(one).\mathtt{end},\ \mathsf{s2} : \mathsf{assert}(two).\mathtt{end}\} \\ S_2 & = & \oplus\{\mathsf{p1} : \mathsf{consume}(one).\mathtt{end},\ \mathsf{p2} : \mathsf{consume}(two).\mathtt{end}\} \end{array}$$

We would like to obtain the following composition:

$$S_{12} = \oplus \left\{ \begin{array}{l} \mathsf{s1} : \oplus\{\mathsf{p1} : \mathsf{assert}(one).\mathsf{consume}(one).\mathtt{end}\}, \\ \mathsf{s2} : \oplus\{\mathsf{p2} : \mathsf{assert}(two).\mathsf{consume}(two).\mathtt{end}\} \end{array} \right\}$$

[1] If we start from a non-empty environment $\{n\}$ we can derive $?\mathtt{pwd}.\oplus\{\mathsf{ok} : \mathsf{assert}(n).\mathsf{require}(n).S', \mathsf{ko} : \mathsf{require}(n).S'\}$. However, initial assumption $\{n\}$ means that access to S_B is granted regardless of the authentication outcome.

Composition rule [bra] is too strict and returns an empty set for S_1 and S_2. Weak branching [wbra] is also not useful in this case, producing the interleaving below, which does not capture the intended correlation:

$$\oplus\left\{\begin{array}{l}\mathsf{p1} : \oplus\left\{\ \mathsf{s1} : \mathsf{assert}(\mathit{one}).\mathsf{consume}(\mathit{one}).\mathsf{end}, \mathsf{s2} : \mathsf{assert}(\mathit{two}).\mathsf{end}\ \right\}, \\ \mathsf{p2} : \oplus\left\{\ \mathsf{s1} : \mathsf{assert}(\mathit{one}).\mathsf{end}, \mathsf{s2} : \mathsf{assert}(\mathit{two}).\mathsf{consume}(\mathit{two}).\mathsf{end}\ \right\}\end{array}\right\}$$

Definition 7 introduces a further rule [cbra], to allow for correlating compositions.

Definition 7 (Correlating branching) Correlating branching composition *is derived using the judgement in Definition 5 with the addition of rule [cbra] below:*

$$\frac{\begin{array}{c}\forall i \in I.\, J_i \neq \emptyset\ \wedge \bigcup_{i\in I} J_i = J \\ \forall j \in J_i \quad T_L; T_R; A \vdash S_i \circ S'_j \triangleright S_{ij} \qquad\qquad \forall j \in J \setminus J_i \quad T_L; T_R; A \vdash S_i \circ S'_j \not\triangleright\end{array}}{T_L; T_R; A \vdash +\{\mathsf{l}_i : S_i\}_{i\in I} \circ +'\{\mathsf{l}'_j : S'_j\}_{j\in J} \triangleright +\{\mathsf{l}_i : +'\{\mathsf{l}'_j : S_{ij}\}_{j\in J_i}\}_{i\in I}}$$

The first premise requires that: (1) each branch of the first protocol can be correlated with at least one branch of the second protocol ($J_i \neq \emptyset$), and (2) each branch of the second protocol can be correlated with at least one branch of the first protocol ($\bigcup_{i\in I} J_i = J$). This precondition is critical to ensure the fairness property we introduce in Section 4 (Definition 9). Rule [cbra] allows us to obtain S_{12} as the interleaving composition of S_1 and S_2 above, modelling the intended correlation.

Hereafter we denote with $\triangleright_c$ (resp. $\triangleright_{wc}$) derivations obtained using the judgements in Definition 7 with the addition of rule [cbra] (resp. [cbra] and [wbra]). The inclusion relation between the different kinds of judgement is shown on the right (with $\triangleright_s$ and $\triangleright_{wc}$ being the most and least strict, respectively).

$$\begin{tikzcd}
 & \triangleright_w \arrow[dr, phantom, "\subset"] & \\
\triangleright_s \arrow[ur, phantom, "\subset"] \arrow[dr, phantom, "\subset"] & & \triangleright_{wc} \\
 & \triangleright_c \arrow[ur, phantom, "\subset"] &
\end{tikzcd}$$

4 Properties of interleaving composition

In this section, we give the main properties of interleaving compositions. First, we give some general properties of well-assertedness and algebraic/scoping properties (i.e., sanity checks). Then, we give behaviour preservation and fairness, both formulated using a semantics of 'protocol ensembles' (a semantic counterpart of syntactic composition). Hereafter, we will denote with $\triangleright$ any kind of judgement in $\{\triangleright_s, \triangleright_w, \triangleright_c, \triangleright_{wc}\}$.

4.1 Well-assertedness of compositions

Critical for the validity of our approach is that interleaving compositions preserve the constraints of assertions:

Proposition 3 (Validity) *If $T_L; T_R; \emptyset \vdash S_1 \circ S_2 \triangleright S$ then S is very-well-asserted.*

?? details the proof. A corollary of Proposition 3 and Lemma 3 (progress of very-well-asserted protocols) is that interleaving compositions enjoy progress:

Corollary 1 (Progress) *If $T_L; T_R; \emptyset \vdash S_1 \circ S_2 \triangleright S$ then S enjoys progress.*

4.2 Algebraic and scoping properties

We consider algebraic properties and notions of open and closed protocol with respect to recursion variables. Appendix **??** details the proofs of these results.

Composing closed recursive protocols yields closed protocols. This property is a corollary of a more general property, that free variables are preserved by composition:

Proposition 4 *If* $T_L; T_R; A \vdash S_1 \circ S_2 \triangleright S$ *then* $\mathsf{fv}(S_1) \cup \mathsf{fv}(S_2) = \mathsf{fv}(S)$.

That is, the free variables of a composed protocol are exactly the union of the free variables of the protocols being composed.

Corollary 2 (Composition preserves closedness) *For all* A, S *and closed protocols* S_1, S_2*, if* $T_L; T_R; A \vdash S_1 \circ S_2 \triangleright S$ *then* S *is a closed protocol.*

A useful algebraic property is that composition has end protocols as units:

Proposition 5 (Interleaving composition has left- and right-units) *For a protocol* S *where* $A\ \{S\} \wedge \mathsf{fv}(S) = \emptyset$ *then* $T_L; T_R; A \vdash S \circ \mathtt{end} \triangleright S$ *and* $T_L; T_R; A \vdash \mathtt{end} \circ S \triangleright S$.

4.3 Behaviour preservation and fairness of protocol ensembles

In Section 3, we gave a syntactic definition of *interleaving composition*, which enacts the dependencies implied by assertions in protocols, and provides a blue-print of an implementation. In this section, we consider 'protocol ensembles', which can be understood as the *semantic compositions* of two asserted protocols. Semantic compositions have a behaviour that is similar to parallel composition (e.g., as in CCS), but unlike parallel composition the two asserted protocols cannot communicate with each other, i.e., there are no internal τ actions. All interactions in a semantic composition are directed towards other endpoints. Semantic composition provides a more general and somewhat familiar notion of composition, which we will use as a reference to analyse the properties of interleaving compositions.

Protocol ensembles, ranged over by C, are defined as follows:

$$C \quad ::= \quad S \quad | \quad S \,||\, S$$

By defining C as either asserted protocols S (which may be interleaving compositions) or semantic compositions $S \,||\, S$, we obtain a common LTS for comparing the behaviour of interleaving and semantic compositions. For simplicity we limit the theory to the composition of two protocols. The extension to n protocols is straightforward e.g., based on labelling each protocol and its actions with a unique identifier.

The LTS for protocol ensembles extends the LTS for asserted protocols: it is defined over states of the form (A, C), transition labels ℓ (as for asserted protocols), and by the rules in Definition 2 plus the following two rules:

$$\frac{(A, S_1) \xrightarrow{\ell} (A', S_1')}{(A, S_1 \,||\, S_2) \xrightarrow{\ell} (A', S_1' \,||\, S_2)}\ \langle\mathtt{Com1}\rangle \qquad \frac{(A, S_2) \xrightarrow{\ell} (A', S_2')}{(A, S_1 \,||\, S_2) \xrightarrow{\ell} (A', S_1 \,||\, S_2')}\ \langle\mathtt{Com2}\rangle$$

We write $(A, C) \rightarrow$ if $(A, C) \xrightarrow{\ell} (A', C')$ for some ℓ, A', C'. Protocols in C do not communicate internally, but may affect each other by changing or checking A.

Behaviour preservation Fix an LTS for protocol ensembles $(Q, L, \rightarrow)$ defined on the set Q of states $\mathbf{s}$ of the form (A, C) and labels L. We use the standard notion of *simulation* [39] to compare protocols of interleaving compositions and protocol ensembles, using protocol ensembles as a correct general model to which interleaving compositions need to adhere.

Definition 8 (Simulation) *A (strong)* simulation *is a relation $\mathcal{R} \subseteq Q \times Q$ such that, whenever $\mathbf{s}_1 \mathcal{R} \mathbf{s}_2$: $\forall \ell \in L, \mathbf{s}'_1 : \mathbf{s}_1 \xrightarrow{\ell} \mathbf{s}'_1$ implies $\exists \mathbf{s}'_2 : \mathbf{s}_2 \xrightarrow{\ell} \mathbf{s}'_2$ and $\mathbf{s}'_1 \mathcal{R} \mathbf{s}'_2$.*

We call 'similarity' the largest simulation relation. We write $\mathbf{s}_1 \lesssim \mathbf{s}_2$ when there exists a simulation $\mathcal{R}$ such that $\mathbf{s}_1 \mathcal{R} \mathbf{s}_2$. We say that C_1 preserves the behaviour of C_2 with respect to A if $(A, C_1) \lesssim (A, C_2)$.

Theorem 1 (Behaviour preservation of compositions - closed)

$$\emptyset; \emptyset; A \vdash S_1 \circ S_2 \triangleright S \quad \Rightarrow \quad (A, S) \lesssim (A, S_1 \,||\, S_2)$$

Therefore, interleaving compositions will only show behaviour that would be allowed by a protocol ensemble. Clearly, protocol ensembles allow more possible executions than an interleaving composition, which is only one of the possible interleavings. The proof of Theorem 1 is by induction on the derivation of S and, although the statement assumes closed protocols, some inductive hypotheses in the proof (e.g., premises of [rec1] or [rec2]) require reasoning about open protocols. The proof hence relies on a property (Lemma **??** – Appendix **??**) on open protocols: (roughly) given two protocols and one of their interleaving compositions, any action of the interleaving composition is matched by an action of the ensemble of the two protocols, and this property is preserved upon transition. Note that, while environments T_L and T_R are trivially empty in Theorem 1 (closed protocols), they have a key role in proving **??** (open protocols): they include the variables of each component protocol that have been bound in a derivation, and give critical information of the scope and structure of the original component protocols in that derivation. **??** details the proof.

Fairness Fix an ensemble of two protocols $S_0 \,||\, S_1$ and any of their interleaving compositions S. By fairness, each action of S_0 (resp. S_1) can be observed in at least one execution of S, possibly after a finite sequence of other actions by S_1 (resp. S_0). In the following, we write $(\_, S)$ to denote (A, S) when A is immaterial.

Definition 9 (Fairness) *S is* fair *w.r.t. S_0 and S_1 on A, if $\forall i \in \{0, 1\}$ and any transition $(\_, S_i) \xrightarrow{\ell} (\_, S'_i)$ there exists $\vec{r}$ such that:* ***1)*** *$(A, S_{|1-i|}) \xrightarrow{\vec{r}} (\_, S'_{|1-i|})$,* ***2)*** *$(A, S) \xrightarrow{\vec{r}\ell} (A', S')$, and* ***3)*** *$S'$ is fair with respect to S'_i and $S'_{|1-i|}$ on A'.*

Theorem 2 (Fairness of compositions) *If $\emptyset; \emptyset; A \vdash S_0 \circ S_1 \triangleright S$ then S is* fair *w.r.t. S_0 and S_1 on A.*

A key aspect of fairness (Definition 9) is that it fixes ℓ and then requires at least one execution in which ℓ is eventually executed by S. This implies that although not all possible future branches include all parts of the protocols being composed, some will.

Definition 10 (Strong fairness) *S is strongly fair w.r.t. S_0 and S_1 on A, if any $i \in \{0,1\}$ and all transitions $(\_, S_i) \xrightarrow{\ell} (\_, S_i')$ and $(A, S_{|1-i|}) \xrightarrow{\vec{r}}$, there exist $\vec{r}'$, $\vec{r}''$ with $(A, S_{|1-i|}) \xrightarrow{\vec{r}'} (\_, S'_{|1-i|})$ and either:*

1) $\vec{r}'\vec{r}'' = \vec{r}$ (i.e., $\vec{r}'$ is a prefix of $\vec{r}$), or

2) $\vec{r}' = \vec{r}\vec{r}''$ (i.e., $\vec{r}$ is an ex prefix of $\vec{r}'$)

such that $(A, S) \xrightarrow{\vec{r}'\ell} (A', S')$ and S' is strongly fair w.r.t. S_i' and $S'_{|1-i|}$ on A'.

By Definition 10, any action of a composition can be matched by an action of the protocols being composed, and this property is preserved by transition. Vectors $\vec{r}$, $\vec{r}'$, and $\vec{r}''$ are used to universally quantify on $\vec{r}$ and yet allow for the cases where ℓ comes before (1) or after (2) $\vec{r}$ in the composition. It follows a stronger fairness result for compositions using only [bra] that only holds for $\triangleright_s$ judgements.

Theorem 3 (Strong fairness of compositions with $\triangleright_s$) *If $\emptyset; \emptyset; A \vdash S_0 \circ S_1 \triangleright_s S$ then S is* strongly fair *with respect to S_0 and S_1 on A.*

Appendix **??** details the proofs.

Example 3 (Fairness and weak branching) *Consider a simpler variant of the protocols in Section 3.1.1 (omitting password exchange and continuation):*

$$\begin{array}{lcl} S_A & = & \oplus\{\mathsf{ok} : \mathsf{assert}(n).\mathsf{end},\ \mathsf{ko} : \mathsf{end}\} \qquad S_B = \mathsf{require}(n).\ \mathsf{end} \\ S_{AB} & = & \oplus\{\mathsf{ok} : \mathsf{assert}(n).\mathsf{require}(n).\mathsf{end},\ \mathsf{ko} : \mathsf{end}\} \end{array}$$

Observe $\emptyset; \emptyset; \emptyset \vdash S_A \circ S_B \not\triangleright_s S_{AB}$ and $\emptyset; \emptyset; \emptyset \vdash S_A \circ S_B \triangleright_w S_{AB}$. We show that S_{AB} is a fair composition w.r.t. S_A and S_B on $\emptyset$, but it is not a strongly fair one.

First focus on fairness. S_A can move with either label $\oplus\mathsf{ok}$ or $\oplus\mathsf{ko}$. In either case $(\emptyset, S_{AB})$ can immediately make a corresponding step with $\vec{r}$ empty. If S_B moves, that is by label $\mathsf{require}(n)$, then for some environment $\{n\}$:

$$(\{n\}, S_B) \xrightarrow{\mathsf{require}(n)} (\emptyset, \mathsf{end}) \tag{1}$$

There exists a sequence of transitions with labels $\vec{r} = \oplus\mathsf{ok}, \mathsf{assert}(n)$ such that

$$(\emptyset, S_B) \xrightarrow{\oplus\mathsf{ok},\mathsf{assert}(n)} (\{n\}, \mathsf{end}) \qquad (\emptyset, S) \xrightarrow{\oplus\mathsf{ok},\mathsf{assert}(n)} (\{n\}, \mathsf{require}(n).\mathsf{end}) \xrightarrow{\mathsf{require}(n)} (\emptyset, \mathsf{end})$$

and $\emptyset; \emptyset; \emptyset \vdash \mathsf{end} \circ \mathsf{end} \triangleright_w \mathsf{end}$. In the case above, we could select a 'good' path of S_A and S_{AB} that allows the transition with label $\mathsf{require}(n)$ to happen.

Focus now on strong fairness and again, consider the step in Equation (1) by S_B. Now we can pick an arbitrary $\vec{r}$, say, $\oplus\mathsf{ok}$, such that $(\emptyset, S_B) \xrightarrow{\oplus\mathsf{ko}} (\emptyset, \mathsf{end})$. Looking at S_{AB}, there is no prefix nor extension of $\vec{r} = \oplus\mathsf{ok}$ that allows a $\mathsf{require}(n)$ step by S_{AB} once the branch ko is taken. Therefore, S_{AB} is not strongly fair with respect to S_A and S_B on $\emptyset$.

4.4 Completeness

We discuss completeness of our composition rules: for every ‘good’ execution of $S_1 \,||\, S_2$ (i.e., non-terminating or reaching state $\mathtt{end} \,||\, \mathtt{end}$), can we obtain an interleaving composition of S_1 and S_2 that yields that execution? At present the answer is negative. For example, S_a and S_b below produce no interleavings (not even with $\triangleright_w$)

$$\begin{aligned} S_a &= \mathtt{?pwd}.\mathsf{assert}(login).\mathtt{?quit}.\mathsf{assert}(n).\mathsf{consume}(login).\mathtt{end} \\ S_b &= \mu\mathtt{t}.\&\{\mathsf{balance} : \mathsf{require}(login).\mathtt{!bal}.\mathtt{t},\ \mathsf{finish} : \mathsf{consume}(n).\mathtt{end}\} \end{aligned}$$

while it may be desirable to obtain:

$$\mathtt{?pwd}.\mathsf{assert}(login).\mu\mathtt{t}.\&\begin{cases} \mathsf{balance} : \mathsf{require}(login).\mathtt{!bal}.\mathtt{t}, \\ \mathsf{finish} : \mathtt{?quit}.\mathsf{assert}(n).\mathsf{consume}(n).\mathsf{consume}(login).\mathtt{end} \end{cases}$$

The IC above cannot be derived because [rec1] prevents composition of recursive with non-recursive protocols. A simplistic modification of [rec1] to allow composition of $\mathtt{t}_1.S_1$ and S_2 (with $\mathtt{Top}(S_2) = \emptyset$) would produce $\mu\mathtt{t}.\mathtt{?pwd}.\mathsf{assert}(login).\&\{\ldots\}$ which is not behaviour preserving (the password request is repeated). Similar tweaks to [rec2] have the same problem. With more complex rules, we may possibly allow weak composition of S_a with S_b only for syntactic subterms of S_b that terminate (e.g, after the finish branch). Extending our rules in this direction, and investigating completeness, is future work. At present using $\triangleright_c$ we can still compose S_b with a modified S_a, e.g.

$$\mathtt{?pwd}.\mathsf{assert}(login).\mu\mathtt{t}_a.\&\{\mathsf{void} : \mathtt{t}_a, \mathsf{quit} : \mathtt{?quit}.\mathsf{assert}(n).\mathsf{consume}(login).\mathtt{end}\}$$

5 Implementation

To illustrate the proposed approach, we have implemented a tool for Erlang that offers *interleaving composition of protocols*, *code generation*, and *protocol extraction*.

Interleaving composition is defined as a function producing zero or more protocol compositions, giving an algorithmic implementation of the relation in Definition 5. Following the variations on the branching rule, the tool offers strong, weak, correlating, and weak/correlating (denoted All in the table) composition. The user can select the kind of branching they wish to use. Looking at Example 1, the strong composition of banking and authentication protocols returns an empty set as expected. When opting for weak composition instead, the tool outputs one IC, equivalent to Example 2:

Listing 1 PIN/TAN Banking Protocol rendered in our Erlang AST for protocols

```
bank_pintan() ->
  {act,r_pin, {branch, [{ok, {assert, pin, {require, pin, {rec, "r",
                                  {branch, [{payment, {assert, pay, {consume, pay, {act,s_id, {act,r_tan,
                                     {branch, [{ok, {assert, tan, {consume, tan, {act, r_details, {rvar, "r"}}}}},
                                              {fail, {rvar, r}}]}}}}}}},
                                              {statement, {act, s_statement, {rvar, "r"}}},
                                              {logout, {consume, pin, endP}}]}}}}},
                            {fail, endP}]}}
```

Table 1 Number of compositions for branching rule variations; running example in grey.

№	Protocols	Strong	Weak	Correlating	All
1	service(), login()	0	1	0	1
2	s1(), s2() (Section 3.1.2)	0	1	2	3
3	i1(), i2() (Section 2.1)	1	1	1	1
4	http(), aws_auth() (from [26])	0	6	0	6
5	login(), booking()	0	1	0	1
6	pin(), tan()	0	1	0	1
7	pintan(), bank()	0	1	0	1
8	resource(), server()	1	1	1	2
9	userAgent(), agentInstrument()	0	0	2	2
10	bankauthsimple(), keycard()	0	1	0	1
11	auth_two_step(), email()	0	9	0	9
12	sa(), sb() (Section 4.4)	0	12	2	14

Offering all four composition options (corresponding to $\triangleright_s$, $\triangleright_w$, $\triangleright_c$, $\triangleright_{wc}$ in the theory) instead of offering only the less restrictive weak/correlating branching $\triangleright_{wc}$, may improve the relevance of compositions returned. As observed in Section 3.1.2, using [wbra] in a context where we need to correlate branches likely returns irrelevant compositions (e.g. row 12). One way to reduce the number of irrelevant compositions, is to introduce more assertions. In fact, one of the aims of the tool is to support step-wise understanding of the protocol via progressive use of assertions. An alternative would be to annotate branching instances with the different options, which would further increase relevance of the returned results. This is left for future work. Table 1 shows the number of interleaving compositions obtained for each variation of the branching rule for a suite of examples. The suite includes: ad-hoc examples to validate the theory (rows 1 - 3, 7, 12), examples from literature, such as the HTTP example from [26] (row 4), and other examples inspired from real-world applications such as Gmail's two-steps authentication (row 11). By appropriately selecting composition options and assertions, the tool returns a small number of interleaving compositions. The number of compositions increases in examples with recursions, especially nested recursions as can be seen in rows 4, 11, which would require some additional assertions to choose among the interleavings.

Code generation takes a protocol definition and produces an Erlang stub. Protocol structures (action, sequence, choice) can be represented as a directed graph and then as finite state machines that transition based on the messages received. The finite state machines are used to generate a stub that uses the Erlang/OTP gen_statem [1], a generic abstraction which supports the implementation of finite state machine modules. Not only is it convenient to represent the protocol as a state machine, but gen_statem offers some useful features. Internal events from the state machine to itself are a good way to represent branches that make a selection among some choices. 'Postponing events' and timeouts provide functionality for further implementation of the generated code stubs. *Actions* and *branches* are represented as events that trigger a state transition. We use function declarations to represent incoming events, and function applications to represent outgoing events. Each state has its own handler function used to send an event to the state machine. When the event is received the corresponding state function is called and the transition to the next state is made. The default generated event is an asynchronous communication (called a 'cast' in Erlang/OTP parlance). For sending actions and selecting branches, the event type is internal, an event from your state machine to itself. *End* is represented by the terminate function of a gen_statem module, whilst the *fixed-point* and the *recursive variables* dictate the control flow of the state machine. State variables must be declared by including them in a record definition — Data. Following Frama-C [15], we represent *assertions* as specially formatted comments. For example: {assert, pay} is represented as an Erlang comment %assert pay. These comments are positioned before code that implements the state to which this assertion acts as a pre-condition in the protocol. Listing 2 shows an excerpt of the code generated for the PIN/TAN Banking protocol, bank_pintan(), containing the states generated for the first action and branch.

Listing 2 PIN/TAN Banking State Machine

```
state1(cast, Pin, Data) -> {next_state, state2, Data}.
%assert pin
%require pin
state2(cast, ok, Data) -> {next_state, state3, Data};
state2(cast, fail, Data) -> {stop, normal, Data}.
%assert pay
%consume pay
state3(cast, payment, Data) -> {next_state, state4, Data};
state3(cast, statement,Data) ->{next_state, state10,Data};
%consume pin
state3(cast, logout, Data) -> {stop, normal, Data}.
```

Protocol extraction and migration. Protocol extraction generates protocols from code via a static analysis of Erlang modules implementing state machines using either gen_statem, or gen_fsm behaviour. When assertions are expressed using the comments illustrated above, they are also extracted. The obtained protocol can be annotated with extra assertions as necessary and composed with another to obtain a more complex protocol. The extraction option preserves local code that can be migrated when generating a new stub. For example, starting out from an existing implementation of banking, we can use the tool to extract the protocol S_B, obtain a composition with S_A, and generate a new module where pre-existing code for banking can be migrated.

Re-engineering. To extend the banking/authentication server with a keycard authentication option, we can compose the PIN/TAN Banking Protocol with e.g., the keycard protocol below. Assertions ensure that the branching for TAN or keycard authentication is plugged in (using assertion keyp) to the payment option of the PIN/TAN protocol, and that TAN authentication in PIN/TAN protocol is plugged only in the tan branch of the keycard protocol (using assertion otp):

Listing 3 Keycard Option Protocol

```
keycard() -> {rec, "y", {require, keyp, {branch, [{tan, {assert, otp, {rvar, "y"}}},
                                                   {keycard, {rvar, "y"}}]}}}.
```

By adding an assertion of keyp and a consume of otp at the beginning of the branch payment of the PIN/TAN Protocol one would obtain the desired extension using the weak composition option. Our tool can then be used to generate a stub for the extended protocol and migrate reusable code from the implementation of the PIN/TAN Banking

Protocol to the new implementation. These features satisfy the requirements laid out in Section 1.1: supporting re-engineering driven by the composition of protocols. The tool generates stubs from ICs, extracts protocols, and reuse code upon composition with different protocols. See our artifact for the complete benchmark [10].

6 Related Work and Conclusion

There is a vast literature on protocol specification (both theory and practice, e.g. see the survey of Lai [34]). Most techniques provide a monolithic view of protocols. We studied protocol composition using 'assertions' to specify contact points and constraints between the protocols. We have given correctness in terms of behaviour preservation, fairness and well-assertedness, and shown that all compositions enjoy it. There are three main lines of research that relate to our work.

Firstly, *software adaptors* give typed protocol interfaces between software components [45]. The idea is similar to the structured view of communication in session types [24], with the notion of *duality* capturing when opposite sides of a protocol are compatible. Composition in these works is about sound composition of protocol implementations, whereas we address the (upfront) creation of composite protocols.

Secondly, protocol composition has been studied as the run-time 'weaving' of component actions. Barbanera et al. study such a composition in the setting of communicating finite state machines [2, 3]. Participants in two communicating systems can be transformed into coupled 'gateways', forming a composite system. A compatibility relation is based on dual behaviour of the two gateways. Safety of the resulting system is by this compatibility, along with conditions of 'no mixed states' and determinism for sends and receives. Building from this idea, later work studies synchronous CFSMs, and replaces the two coupled gateways with a single one [5], and composition/decomposition on global types in the setting of multiparty session types [4]. Montesi and Yoshida study composition in the setting of choreographies [38]. Their composition relies on the use of partial choreographies, which can mix global descriptions with communication among external peers. Inspired by aspect-oriented programming,the work in [41] supports protocol extensions with 'aspectual' session types, that allow messages in session types to be matched and consequently introduce new behaviour in addition to, or in place of, the matched messages. Unlike the above approaches, we focus on a syntactic, statically derivable notion of composition. We use process calculi to model protocols as simple (i.e., mono-thread) objects that can be used by humans to reason about the desired application logic and generate/engineer modular code. The work in [33] looks at composition of aspects and modular verification of aspect-oriented programs, focussing on maintaining a relationship between models and aspects. Various works look at composition of features into coherent software systems [13, 22, 29, 46], focussing on resolving conflict stemming from feature interactions. Instead, we focus on establishing primitives for humans to reason on what a composite protocol should be, and support code generation.

The third pertinent thread in the literature defines syntactic compositions in the form of Team Automata [18, 44, 43] or related calculi [44]. These works define different ways of composing machines, primarily based on synchronising machines via common actions. In contrast, our means of composition is via assertions (orthogonal to actions) which express directional (i.e., rely-guarantee-style) dependencies. Our use of assertions aims to reflect programming practice. Assertions are kept in our generated code and can be used to enable protocol extraction and re-engineering, and as code documentation. Our composition cannot capture the whole range of synchronisations offered by Team Automata. Conversely, Team Automata cannot capture the range of compositions in our approach. One can encode some interleaving compositions as Team Automata, by modelling each assert(n)-require(n) or assert(n)-consume(n) pair as a common synchronisation action. However, the options offered by Team Automata (e.g., 'free', 'state indispensable', or 'action indispensable') do not capture our requirement that synchronisation (i) always happens on assertion-actions and (ii) never happens on communication actions (these are a separate syntactic and semantic entity). Furthermore, our assertions do not imply immediate synchronisation: an assert(n) can occur in a protocol some way before a require(n). Thus an attempted encoding of Team Automata into our protocols, encoding synchronisation actions as unique assert(n)-consume(n) pairs, would not preserve the behaviour of Team Automata for all possible compositions (just those where 'annihilating' pairs appeared contiguously). Thus, Team Automata and our approach overlap in some synchronising behaviours, but not all. A formal study of such overlap is further work.

Unlike approaches to safe communication discussed above, we do not focus on communication safety, which is an orthogonal concern. As discussed in **??**, our parameterisable language allows us to inherit communication-safety properties from session types by instantiating our protocol language to a session type syntax (e.g. that in [16]), with asynchrony [14] and multiparty sessions [8].

We are working on a factorisation function that decomposes protocols, as a kind of algebraic inverse to composition. This would allow us to 'close the loop', factorizing protocols into simple components for later (re)composition. We plan to extend recursion to model quantified recursion and assertion environments as multisets.

Acknowledgements This work has been partially supported by the BehAPI project funded by the EU H2020 RISE under the Marie Sklodowska-Curie action (No: 778233), EPSRC project EP/T014512/1 (STARDUST), and EPSRC project EP/T013516/1 (Granule). We thank Christian Kissig for contributing with ideas and discussions, and the anonymous reviewers for their feedback. Orchard is also supported by the generosity of Eric and Wendy Schmidt by recommendation of the Schmidt Futures program.

References

[1] Ericsson AB. Stdlib, reference manual. https://erlang.org/doc/man/gen_statem.html. Last Accessed 21.09.2022.

[2] Franco Barbanera, Ugo de'Liguoro, and Rolf Hennicker. Global types for open systems. In Massimo Bartoletti and Sophia Knight, editors, *ICE '18 - 11th Interaction and Concurrency Experience, Madrid, Spain, June 20-21, 2018. Proceedings*, volume 279 of *EPTCS*, pages 4–20, 2018. doi:10.4204/EPTCS.279.4.

[3] Franco Barbanera and Mariangiola Dezani-Ciancaglini. Open multiparty sessions. In Massimo Bartoletti, Ludovic Henrio, Anastasia Mavridou, and Alceste Scalas, editors, *ICE '19 - 12th Interaction and Concurrency Experience, Copenhagen, Denmark, 20-21 June 2019. Proceedings*, volume 304 of *EPTCS*, pages 77–96, 2019. doi:10.4204/EPTCS.304.6.

[4] Franco Barbanera, Mariangiola Dezani-Ciancaglini, Ivan Lanese, and Emilio Tuosto. Composition and decomposition of multiparty sessions. *Journal of Logical and Algebraic Methods in Programming*, 119:100620, 2021. doi:https://doi.org/10.1016/j.jlamp.2020.100620.

[5] Franco Barbanera, Ivan Lanese, and Emilio Tuosto. Composing communicating systems, synchronously. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA '20 - 9th International Symposium on Leveraging Applications of Formal Methods, Rhodes, Greece, October 20-30, 2020. Proceedings, Part I*, volume 12476 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 2020. doi:10.1007/978-3-030-61362-4_3.

[6] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In Franck van Breugel and Marsha Chechik, editors, *CONCUR '08, 19th International Conference on Concurrency Theory, Toronto, Canada, August 19-22, 2008. Proceedings*, volume 5201 of *Lecture Notes in Computer Science*, pages 418–433. Springer, 2008. doi:10.1007/978-3-540-85361-9_33.

[7] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In Franck van Breugel and Marsha Chechik, editors, *CONCUR '08 - 19th International Conference on Concurrency Theory, Toronto, Canada, August 19-22, 2008. Proceedings*, volume 5201 of *Lecture Notes in Computer Science*, pages 418–433. Springer, 2008. doi:10.1007/978-3-540-85361-9_33.

[8] Laura Bocchi, Hernán Melgratti, and Emilio Tuosto. Resolving non-determinism in choreographies. In Zhong Shao, editor, *Programming Languages and Systems*, volume Lecture Notes in Computer Science of *8410*, pages 493–512, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. doi:10.1007/978-3-642-54833-8_26.

[9] Laura Bocchi, Dominic Orchard, and A. Laura Voinea. A theory of composing protocols, August 2022. doi:10.5281/zenodo.7105666.

[10] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In Rocco De Nicola, editor, *ESOP '07 - 16th European Symposium on Programming, Braga, Portugal, March 24 - April 1, 2007.*

Proceedings, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi:10.1007/978-3-540-71316-6_2.

[11] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. *ACM SIGPLAN Notices*, 48(1):263–274, 2013. doi:10.1145/2480359.2429101.

[12] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a feature: A requirements engineering perspective. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4961 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2008. doi:10.1007/978-3-540-78743-3_2.

[13] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Asynchronous session types and progress for object oriented languages. In Marcello M. Bonsangue and Einar Broch Johnsen, editors, *FMOODS '07 - 9th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems, Paphos, Cyprus, June 6-8, 2007, Proceedings*, volume 4468 of *Lecture Notes in Computer Science*, pages 1–31. Springer, 2007. doi:10.1007/978-3-540-72952-5_1.

[14] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Software Analysis Perspective. In *SEFM '12 - 10th international conference on Software Engineering and Formal Methods, Thessaloniki Greece October 1-5, 2012. Proceedings*, volume 7504 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2012. doi:10.1007/978-3-642-33826-7_16.

[15] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017. doi:10.1016/j.ic.2017.06.002.

[16] Clarence Ellis. Team automata for groupware systems. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge*, GROUP '97, pages 415–424, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/266838.267363.

[17] Simon Gay and António Ravara. *Behavioural Types: From Theory to Tools*. River Publishers, 2017. doi:10.13052/rp-9788793519817.

[18] Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005. doi:10.1007/s00236-005-0177-z.

[19] Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele, editors. *Abstract State Machines, Theory and Applications, International Workshop, ASM 2000, Monte Verità, Switzerland, March 19-24, 2000, Proceedings*, volume 1912 of *Lecture Notes in Computer Science*. Springer, 2000. doi:10.1007/3-540-44518-8.

[20] Jonathan D. Hay and Joanne M. Atlee. Composing features and resolving interactions. In John C. Knight and David S. Rosenblum, editors, *ACM SIGSOFT Symposium on Foundations of Software Engineering, an Diego, California, USA, November 6-10, 2000, Proceedings*, pages 110–119. ACM, 2000. doi:10.1145/355045.355061.

[21] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

[22] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *ESOP '98, 7th European Symposium on Programming, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

[23] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *POPL '08, 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 7-12, 2008. Proceedings*, pages 273–284. ACM, 2008. doi:10.1145/1328438.1328472.

[24] Raymond Hu. Distributed Programming Using Java APIs Generated from Session Types. *Behavioural Types: from Theory to Tools*, pages 287–308, 2017. doi:10.13052/rp-9788793519817.

[25] Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint api generation. In Perdita Stevens and Andrzej Wąsowski, editors, *Fundamental Approaches to Software Engineering*, pages 401–418, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. doi:10.1007/978-3-662-49665-7_24.

[26] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.

[27] Michael Jackson and Pamela Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Trans. Software Eng.*, 24(10):831–847, 1998. doi:10.1109/32.729683.

[28] John B Kam and Jeffrey D Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977. doi:10.1007/BF00290339.

[29] Joachim Kock. Note on commutativity in double semigroups and two-fold monoidal categories. *arXiv preprint math/0608452*, 2006.

[30] Dexter Kozen. On kleene algebras and closed semirings. In *MFCS '90 - International Symposium on Mathematical Foundations of Computer Science, Banska Bystrica, Czechoslovakia August 27-31, 1990. Proceedings*, pages 26–47. Springer, 1990. doi:10.1007/BFb0029594.

[31] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. In Richard N. Taylor and Matthew B. Dwyer, editors, *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004*, pages 137–146. ACM, 2004. doi:10.1145/1029894.1029916.

[32] Richard Lai. A survey of communication protocol testing. *Journal of Systems and Software*, 62(1):21–46, 2002. doi:10.1016/S0164-1212(01)00132-7.

[33] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. doi:10.1007/3-540-10235-3.

[34] Fabrizio Montesi. *Choreographic Programming*. PhD thesis, Denmark, 2014.

[35] Fabrizio Montesi and Nobuko Yoshida. Compositional choreographies. In Pedro R. D'Argenio and Hernán C. Melgratti, editors, *CONCUR '13 - 24th International Conference on Concurrency Theory, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, volume 8052 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2013. doi:10.1007/978-3-642-40184-8_30.

[36] Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

[37] Robert E Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, (1):157–171, 1986. doi:10.1109/TSE.1986.6312929.

[38] Nicolas Tabareau, Mario Südholt, and Éric Tanter. Aspectual session types. In Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld, editors, *MODULARITY '14 -13th International Conference on Modularity, Lugano, Switzerland, April 22-26, 2014. Proceedings*, pages 193–204. ACM, 2014. doi:10.1145/2577080.2577085.

[39] Maurice H. ter Beek, Rolf Hennicker, and Jetty Kleijn. Team automata@work: On safe communication. In Simon Bliudze and Laura Bocchi, editors, *COORDINATION '20 - 22nd IFIP WG 6.1 International Conference on Coordination Models and Languages, Valletta, Malta, June 15–19, 2020. Proceedings*, volume 12134 of *Lecture Notes in Computer Science*, pages 77–85, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-50029-0.

[40] Maurice H. ter Beek and Jetty Kleijn. Team automata satisfying compositionality. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME '03: International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003. Proceedings*, volume 2805 of *Lecture Notes in Computer Science*, pages 381–400, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. doi:10.1007/978-3-540-45236-2_22.

[41] Daniel M Yellin and Robert E Strom. Protocol specifications and component adaptors. *TOPLAS '97 - ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997. doi:10.1145/244795.244801.

[42] Pamela Zave and Michael Jackson. New feature interactions in mobile and multimedia telecommunications services. In Muffy Calder and Evan H. Magill, editors, *Feature Interactions in Telecommunications and Software Systems VI, May 17-19, 2000, Glasgow, Scotland, UK*, pages 51–66. IOS Press, 2000.

About the authors

Laura Bocchi l.bocchi@kent.ac.uk.

Dominic Orchard d.a.orchard@kent.ac.uk.

A. Laura Voinea laura.voinea@glasgow.ac.uk.