The Paths to Choreography Extraction*

Luís Cruz-Filipe, Kim S. Larsen, and Fabrizio Montesi

University of Southern Denmark {lcf,kslarsen,fmontesi}@imada.sdu.dk

Abstract. Choreographies are global descriptions of interactions among concurrent components, most notably used in the settings of verification and synthesis of correct-by-construction software. They require a top-down approach: programmers first write choreographies, and then use them to verify or synthesize their programs. However, most software does not come with choreographies yet, which prevents their application. To attack this problem, previous work investigated choreography extraction, which automatically constructs a choreography that describes the behavior of a given set of programs or protocol specifications.

We propose a new extraction methodology that improves on the state of the art: we can deal with programs that are equipped with state and internal computation; time complexity is dramatically better; and we capture programs that work by exploiting asynchronous communication.

1 Introduction

Choreographies are global descriptions of interactions among components. They have been used as a basis for different models and tools that aim at tackling the complexity of modern software, where separate units – such as processes, objects, and threads – interact to reach a common goal [3, 25].

Two lines of research are of particular interest. In *choreography specifications*, choreographies specify interaction protocols, e.g., multiparty session types [17]. In *choreographic programming* [20], choreographies are programs that define the behavior of concurrent algorithms [13] and/or distributed systems [5, 6, 14]. The key feature of these works is EndPoint Projection (EPP), a procedure that translates choreographies to correct endpoint behaviors in lower-level models. For choreography specifications, EPP generates the local specifications of each participant; these specifications can then be used for verification, to check whether some programs implement their role in the protocol correctly and will thus interact without problems at runtime [17]. In choreographic programming, instead, EPP generates correct-by-construction implementations in a model of executable code (program synthesis), typically given in terms of a process calculus [6].

EPP implements a top-down development methodology: developers first write choreographies and then use the output mechanically generated by EPP. However, there are scenarios where this methodology is not applicable; for example:

* Montesi was supported by CRC (Choreographies for Reliable and efficient Communication software), grant DFF–4005-00304 from the Danish Council for Independent Research. Cruz-Filipe and Larsen were supported in part by the Danish Council for Independent Research, Natural Sciences, grant DFF-1323-00247.

– Analysis or integration of legacy software: either code developed previously, or new code written in a technology without support for choreographies.
– Updates: endpoint programs generated by EPP can later be updated locally (e.g., for configuration or optimizations). Since the original choreography is not automatically updated, rerunning EPP loses these changes.

To attack these issues, previous work investigated a procedure to infer choreographies from arbitrary endpoint descriptions. We call this procedure *choreography extraction*. To the best of our knowledge, the current reference for extracting choreography specifications is [19], where graphical choreographies that represent protocol specifications are extracted from communicating automata [4]. Instead, the state of the art for extraction in choreographic programming is [7], where extraction takes terminating processes typed using a fragment of linear logic as input. We advance both lines of work in several aspects, described below.

1.1 Contributions

Extraction for synchronous systems. We define an extraction procedure that applies directly to both choreography specifications and choreographic programming, by working with representative models. We focus on the more difficult case of choreographic programming, and then show how our approach can be applied to other settings in § 6. First we define an extraction algorithm for processes with synchronous communications (§ 4), which showcases the key elements of our construction: building a choreography corresponds to finding paths in a graph that represents the abstract execution of the input processes. Our extraction also helps in debugging: if extraction detects a potential deadlock, we pinpoint it with a special term (**1**). This is the first extraction procedure for choreographic programming that can deal with procedures and infinite behavior [7].

Asynchrony. We extend our development to asynchronous communication (§ 5). The key novelty is that we can extract a new class of behaviors where processes progress because of asynchronous communication. The simplest example of this class is a two-way exchange: a network of two processes where each process starts by sending a value to the other, and then consumes the received value. This network is deadlocked under a synchronous semantics, violating the state-of-the-art requirements for extraction [19]. Capturing these behaviors is challenging for two reasons: there is no choreography language capable of representing them; and the extraction algorithms presented so far require the behaviors of processes to be representable also under a synchronous interpretation. We overcome both limitations with a new choreography primitive for multiparty asynchronous exchange and a look-ahead mechanism for asynchronous actions in extraction.

Efficiency. We show that our extraction has exponential worst-case time complexity in both the synchronous and the asynchronous cases (§ 4 and § 5, respectively), unlike the factorial case of [19], even though we can capture a new class of behaviors. In particular, we need only one phase of exponential complexity, while [19] uses multiple phases applied in sequence. The authors of [19]

detail only the complexities of their first two phases: the first has exponential complexity (but in a quantity larger than ours), while the second has factorial complexity in a function exponential in the size of the input. Our better time complexity stems from the design of our process language, which does not allow non-deterministic receives from different channels, and careful algorithm crafting. Despite the restriction, we can still model interesting examples thanks to asynchronous exchange. In § 5, we present a novel formulation of the alternating 2-bit protocol, which is given in [15] and used in [19] as a motivating example. Our formulation is simpler and does not require threads as in [19].

2 Related Work

Choreographic Programming. The state of the art for extraction in choreographic programming is [7], where synchronous processes with finite behavior are typed using the multiplicative-additive fragment of linear logic. Our approach is significantly more expressive, bringing support for recursion and asynchronous communication. Also, the proof theory in [7] requires that there are no cycles in the structure of connections among processes. We do not have this limitation.

Choreography Specifications. To the best of our knowledge, the state of the art for extracting choreography specifications is [19], which captures more behaviors than previous works with similar objectives [18, 22].

Extraction in [19] is more restrictive wrt. to asynchrony, requiring all process traces and choices to be represented in the synchronous transition system of the network. Thus, networks that are safe because of asynchronous communication are not extracted in [19]. Instead, our extraction can deal with programs that use multiparty asynchronous exchange, where multiple processes exchange values by exploiting asynchronous communication. As a consequence, we can extract the alternating 2-bit protocol implemented via asynchronous exchange in § 5, which is deadlocked under a synchronous semantics and thus cannot be extracted in [19]. Our extraction is the first capturing systems that are not correctly approximated by synchronous semantics (cf. [2]). A precise characterization of the class of extractable systems is thus an interesting future direction.

To circumvent the limitation that asynchronous exchange is not supported, choreographies in [19] support local concurrency: processes can have internal threads. This opens up for an alternative formulation of the alternating 2-bit protocol, where the two participants use two threads each. However, these choreographies are harder to read. As an example, compare our choreography for the alternating 2-bit protocol in § 5 to that obtained with the automata in [19] (given in [15], Protocol 7 in Example 2.1). Our formulation is a simple recursive procedure with two exchanges, whereas the control flow in [15] is rather intricate and uses three different operators (fork, join, and merge) at different places to compose two separate loops. In our opinion, our choreographies follow the principles of structured programming to a greater extent, and are simpler; also because coordination happens only through communication.

$$C ::= \mathbf{0} \mid \eta; C \mid \mathsf{if}\ \mathsf{p} \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{def}\ X = C_2\ \mathsf{in}\ C_1 \mid X$$

$$\eta ::= \mathsf{p}.e\ \texttt{->}\ \mathsf{q} \mid \mathsf{p}\ \texttt{->}\ \mathsf{q}[l] \qquad\qquad e ::= v \mid * \mid \ldots$$

Fig. 1. Core Choreographies, Syntax.

More interestingly than readability, local concurrency makes the complexity of extraction blow up factorially [19]: process threads are represented using non-determinism between different actions in communicating automata. Determining whether the non-deterministic behavior of these automata is extractable takes (super-)factorial time (factorial time in the size of a graph similar to our AES, cf. Definition 2)! Thus, asynchronous exchange supports a more efficient way of capturing an interesting class of behaviors. Nevertheless, we believe that developing efficient extractions of local concurrency may be useful future work.

3 Core Choreographies and Stateful Processes

We review the languages of Core Choreographies (CC) and Stateful Processes (SP), from [11], which respectively model choreographies and endpoint programs. We introduce labels in the reduction semantics for these calculi to formalize the link between choreographies and their process implementations as a bisimilarity.

Core Choreographies (CC). The syntax of CC is given in Figure 1. A choreography C describes the behavior of a set of processes (p, q, ...) running concurrently. Each process has an internal memory cell storing a local value (the value of the process). Term $\mathbf{0}$ is the terminated choreography (omitted in examples). Term $\eta; C$ reads "the system executes η and proceeds as C". An interaction η is either: a value communication $\mathsf{p}.e\ \texttt{->}\ \mathsf{q}$, where process p evaluates e and sends the result to process q, which stores it in its memory cell, replacing its previous value; or a selection $\mathsf{p}\ \texttt{->}\ \mathsf{q}[l]$, where p selects l among the branches offered by q. We abstract from the concrete language of expressions e, which models internal computation and is orthogonal to our development, assuming only that: expressions can contain values v and the placeholder $*$, which refers to the value of the process evaluating them; and evaluating expressions always terminates and returns a value. In a conditional $\mathsf{if}\ \mathsf{p} \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$, p checks if its value is equal to q's to decide whether the system proceeds as C_1 or C_2. Term $\mathsf{def}\ X = C_2\ \mathsf{in}\ C_1$ defines a procedure X with body C_2, which can be called in C_1 and C_2 by using term X.

The semantics of CC is given in terms of labeled reductions $C, \sigma \xrightarrow{\lambda} C', \sigma'$; the main reduction rules are given in Figure 2. Reductions are also closed under context (procedure definitions) and under a structural precongruence $\preceq$, allowing procedure calls to be unfolded and non-interfering actions to be executed in any order. The most interesting rule for $\preceq$ is rule $\lfloor$C|Eta-Eta$\rceil$, which swaps communications between disjoint sets of processes (modeling concurrency). The

$$\frac{e[\sigma(\mathsf{p})/*] \downarrow v}{\mathsf{p}.e \;\texttt{->}\; \mathsf{q}; C, \sigma \xrightarrow{\mathsf{p}.v \;\texttt{->}\; \mathsf{q}} C, \sigma[\mathsf{q} \mapsto v]} \lfloor \mathrm{C|Com} \rceil \qquad \frac{}{\mathsf{p} \;\texttt{->}\; \mathsf{q}[l]; C, \sigma \xrightarrow{\mathsf{p} \;\texttt{->}\; \mathsf{q}[l]} C, \sigma} \lfloor \mathrm{C|Sel} \rceil$$

$$\frac{\sigma(\mathsf{p}) = \sigma(\mathsf{q})}{\mathsf{if}\ \mathsf{p} \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \sigma \xrightarrow{\mathsf{p} \stackrel{\leftarrow}{=} \mathsf{q}:\mathsf{then}} C_1, \sigma} \lfloor \mathrm{C|Then} \rceil \qquad \frac{\mathsf{pn}(\eta) \cap \mathsf{pn}(\eta') = \emptyset}{\eta; \eta' \preceq \eta'; \eta} \lfloor \mathrm{C|Eta\text{-}Eta} \rceil$$

Fig. 2. Core Choreographies, Semantics and Structural Precongruence (selected rules).

$$B ::= \mathsf{q}!\langle e \rangle; B \;\mid\; \mathsf{p}?; B \;\mid\; \mathsf{q} \oplus l; B \;\mid \mathsf{p}\&\{l_i : B_i\}_{i \in I} \;\mid\quad N ::= \mathsf{p} \triangleright B \;\mid\; \mathbf{0} \;\mid\; N \,|\, N$$
$$\mid \mathbf{0} \mid \mathsf{if} * \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2 \;\mid\; \mathsf{def}\ X = B_2\ \mathsf{in}\ B_1 \;\mid\; X$$

Fig. 3. Stateful Processes, Syntax.

total function σ maps each process name to the value it stores. Labels λ tell us which action has been performed, which helps stating our later results. In rule $\lfloor$C|Com$\rceil$, v is the value obtained by evaluating ($\downarrow$) the expression e, with $*$ replaced by the value of the sender p, $\sigma(\mathsf{p})$. In the reductum, σ is updated such that the receiver q stores v. Rule $\lfloor$C|Sel$\rceil$ does not alter σ: selections model invoking a method/operation available at the receiver. Rules $\lfloor$C|Then$\rceil$ and $\lfloor$C|Else$\rceil$ (omitted) model conditionals in the standard way. Function $\mathsf{pn}(C)$ returns all the process names that appear in C, and $C \equiv C'$ means $C \preceq C'$ and $C' \preceq C$.

Example 1. We define a simple choreography for client authentication. We write $\mathsf{p} \;\texttt{->}\; \mathsf{c}, \mathsf{s}[l]$ as a shortcut for $\mathsf{p} \;\texttt{->}\; \mathsf{c}[l]; \mathsf{p} \;\texttt{->}\; \mathsf{s}[l]$.

$$\mathsf{def}\ X = \Big(\mathsf{c}.pwd \;\texttt{->}\; \mathsf{a}; \mathsf{if}\ \mathsf{a} \stackrel{\leftarrow}{=} \mathsf{s}\ \mathsf{then}\ (\mathsf{a} \;\texttt{->}\; \mathsf{c}, \mathsf{s}[ok]; \mathsf{s}.t \;\texttt{->}\; \mathsf{c})\ \mathsf{else}\ (\mathsf{a} \;\texttt{->}\; \mathsf{c}, \mathsf{s}[ko]; X)\Big)\ \mathsf{in}\ X$$

In this choreography, a client process c sends a password to an authentication process a, which checks if the password matches that contained in the server-side process s. If the password is correct, a notifies c and s, and s sends an authentication token t to c. Otherwise, a notifies c and s that authentication failed, and a new attempt is made (by recursively invoking X). □

Stateful Processes. The calulus SP models concurrent/distributed implementations. Thus, unlike in CC, actions are now distributed among processes.

The syntax of SP is given in Figure 3. Networks N are parallel compositions of processes $\mathsf{p} \triangleright B$, read "process p has behavior B". An output term $\mathsf{q}!\langle e \rangle; B$ sends the result of evaluating e to q, and then proceeds as B. Outputs are meant to synchronize with input terms at the target process, i.e., $\mathsf{p}?; B$, which receives a value from p to be stored locally and then proceeds as B. Term $\mathsf{q} \oplus l; B$ sends the selection of the branch labeled l to q. Branches are offered by the

$$\frac{e[\sigma(\mathsf{p})/*] \downarrow v}{\mathsf{p} \triangleright \mathsf{q}!\langle e\rangle; B_1 \ \mid\ \mathsf{q} \triangleright \mathsf{p}?; B_2,\ \sigma \xrightarrow{\mathsf{p}.v\ \texttt{->}\ \mathsf{q}} \mathsf{p} \triangleright B_1 \ \mid\ \mathsf{q} \triangleright B_2,\ \sigma[\mathsf{q} \mapsto v]} \lfloor \mathrm{S|Com} \rceil$$

$$\frac{j \in I}{\mathsf{p} \triangleright \mathsf{q} \oplus l_j; B \ \mid\ \mathsf{q} \triangleright \mathsf{p}\&\{l_i : B_i\}_{i \in I},\ \sigma \xrightarrow{\mathsf{p}\ \texttt{->}\ \mathsf{q}[l]} \mathsf{p} \triangleright B \ \mid\ \mathsf{q} \triangleright B_j,\ \sigma} \lfloor \mathrm{S|Sel} \rceil$$

$$\frac{e[\sigma(\mathsf{q})/*] \downarrow \sigma(\mathsf{p})}{\mathsf{p} \triangleright \mathsf{if}\ * \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2 \ \mid\ \mathsf{q} \triangleright \mathsf{p}!\langle e\rangle; B',\ \sigma \xrightarrow{\mathsf{p} \overset{\leftarrow}{=} \mathsf{q}:\mathsf{then}} \mathsf{p} \triangleright B_1 \ \mid\ \mathsf{q} \triangleright B',\ \sigma} \lfloor \mathrm{S|Then} \rceil$$

Fig. 4. Stateful Processes, Semantics (selected rules).

receiver with term $\mathsf{p}\&\{l_i : B_i\}_{i \in I}$, which offers a choice among the labels l_i to p. When one of these labels is selected, the respective behavior B_i is run. Term $\mathsf{if}\ * \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2$ communicates with process q to check whether it stores the same value as the process running this behavior, in order to choose between the continuations B_1 and B_2. Terms $\mathsf{def}\ X = B_2\ \mathsf{in}\ B_1$ and X are procedure definition and call, respectively.

The semantics of SP is given by labeled reductions $N, \sigma \xrightarrow{\lambda} N', \sigma'$, with labels λ as in CC.[1] Figure 4 shows the key rules (see the appendix for the complete set). Two processes can synchronize when they refer to each other. In rule $\lfloor \mathrm{S|Com} \rceil$, an output at p directed at q synchronizes with the dual input action at q – intention to receive from p; in the reductum, q's value is updated. The reduction receives the same label as the equivalent communication term in CC. The other rules shown are similar. The omitted rules are standard, and close the semantics under parallel composition, structural precongruence, and procedure definitions.

Example 2. The following network implements the choreography in Example 1.

$$\begin{aligned}
&\mathsf{c} \triangleright \mathsf{def}\ X = \mathsf{a}!\langle pwd\rangle; \mathsf{a}\&\{ok : \mathsf{s}?,\ ko : X\}\ \ \mathsf{in}\ X \\
\mid\ &\mathsf{a} \triangleright \mathsf{def}\ X = \mathsf{c}?; \mathsf{if}\ * \overset{\leftarrow}{=} \mathsf{s}\ \mathsf{then}\ (\mathsf{c} \oplus ok; \mathsf{s} \oplus ok)\ \mathsf{else}\ (\mathsf{c} \oplus ko; \mathsf{s} \oplus ko; X)\ \ \mathsf{in}\ X \\
\mid\ &\mathsf{s} \triangleright \mathsf{def}\ X = \mathsf{a}!\langle * \rangle; \mathsf{a}\&\{ok : \mathsf{c}!\langle t\rangle,\ ko : X\}\ \ \mathsf{in}\ X
\end{aligned}$$

EndPoint Projection (EPP). As shown in [11], there exists a partial function $[\![\cdot]\!] : \mathrm{CC} \to \mathrm{SP}$, called EndPoint Projection (EPP), that produces correct implementations of choreographies. EPP produces a parallel composition of processes, one for each process name in the original choreography: $[\![C]\!] = \prod_{\mathsf{p} \in \mathsf{pn}(C)} \mathsf{p} \triangleright [\![C]\!]_{\mathsf{p}}$. The rules for computing $[\![C]\!]$ project the local action performed by the process of interest. For example, $[\![\mathsf{p}.e\ \texttt{->}\ \mathsf{q}]\!]_{\mathsf{p}} = \mathsf{q}!\langle e\rangle$ and $[\![\mathsf{p}.e\ \texttt{->}\ \mathsf{q}]\!]_{\mathsf{q}} = \mathsf{p}?$.

The network presented in Example 2 is exactly the EPP of the choreography in Example 1. Observe that the projection of the conditional in the original choreography for the processes c and s is a branching that supports all the

[1] Deviating from [11], we model process values using σ as for CC, for simplicity.

possible choices made by process a in its projected conditional. Producing these branching terms is possible only if, whenever there is a conditional at a process (a in our example), all other processes receive a label that tells them which branch such a process has chosen. (In case the behaviors of the other processes are the same in both cases, producing branching terms is not necessary.) When this cannot be done for a choreography C, the EPP for C is undefined, and we say that C is unprojectable. Conversely, C is *projectable* if $[\![C]\!]$ is defined.

In the remainder, we relate choreographies to network implementations via a strong labeled reduction bisimilarity $\sim$. Bisimilarity is defined as usual [24]: it is the union of all bisimulation relations $\mathcal{R}$, which in our case relate choreographies to networks. A relation $\mathcal{R}$ is one such bisimulation if whenever $C\mathcal{R}N$ we have that, for all σ: i) $C, \sigma \xrightarrow{\lambda} C', \sigma'$ implies $N, \sigma \xrightarrow{\lambda} N', \sigma'$ for some N' with $C'\mathcal{R}N'$; ii) $N, \sigma \xrightarrow{\lambda} N', \sigma'$ implies $C, \sigma \xrightarrow{\lambda} C', \sigma'$ for some C' with $C'\mathcal{R}N'$.

Theorem 1 (adapted from [11]). *If C is projectable, then $C \sim [\![C]\!]$.*

4 Extraction from SP

The finite case. We first investigate *finite SP*, the fragment of SP without recursive definitions, which we use to discuss the intuition behind our extraction.

Definition 1. *We define a rewriting relation $\rightsquigarrow$ on the language of CC extended with terms $(\![N]\!)$, where N is a network in finite SP, as the transitive closure of:*

$$\frac{N \;\equiv\; \mathbf{0}}{(\![N]\!) \rightsquigarrow \mathbf{0}} \qquad \frac{N \;\equiv\; \mathsf{p} \triangleright \mathsf{q}!\langle e\rangle; N_p \mid \mathsf{q} \triangleright \mathsf{p}?; N_q \mid N'}{(\![N]\!) \rightsquigarrow \mathsf{p}.e \;\texttt{->}\; \mathsf{q}; (\![N_p \mid N_q \mid N']\!)}$$

$$\frac{N \;\equiv\; \mathsf{p} \triangleright \mathsf{q} \oplus l_k; N_p \mid \mathsf{q} \triangleright \mathsf{p}\&\{l_1 : N_{q_1}, \ldots, l_n : N_{q_n}\} \mid N'}{(\![N]\!) \rightsquigarrow \mathsf{p} \;\texttt{->}\; \mathsf{q}[l_k]; (\![N_p \mid N_{q_k} \mid N']\!)}$$

$$\frac{N \;\equiv\; \mathsf{p} \triangleright \mathsf{if}\ * \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ N_{p_1}\ \mathsf{else}\ N_{p_2} \mid \mathsf{q} \triangleright \mathsf{p}!\langle e\rangle; N_q \mid N'}{(\![N]\!) \rightsquigarrow \mathsf{if}\ \mathsf{p} \stackrel{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ (\![N_{p_1} \mid N_q \mid N']\!)\ \mathsf{else}\ (\![N_{p_2} \mid N_q \mid N']\!)} \qquad \frac{\textit{no other rule applies}}{(\![N]\!) \rightsquigarrow \mathbf{1}}$$

A network N in finite SP extracts *to a choreography C if $(\![N]\!) \rightsquigarrow C$.*

The last rule guarantees that every network is extractable. Extraction uses structural precongruence (namely, commutativity and associativity of parallel composition) to find matching actions. For finite SP, this is not a problem (the set of networks equivalent to a given one is finite), but it makes extraction nondeterministic, e.g., the network $\mathsf{p} \triangleright \mathsf{q}!\langle e\rangle \mid \mathsf{q} \triangleright \mathsf{p}? \mid \mathsf{r} \triangleright \mathsf{s}!\langle e'\rangle \mid \mathsf{s} \triangleright \mathsf{r}?$ extracts both to $\mathsf{p}.e \;\texttt{->}\; \mathsf{q}; \mathsf{r}.e' \;\texttt{->}\; \mathsf{s}$ and $\mathsf{r}.e' \;\texttt{->}\; \mathsf{s}; \mathsf{p}.e \;\texttt{->}\; \mathsf{q}$. These choreographies are equivalent by Rule ⌊C|Eta-Eta⌉ (Figure 2). This holds in general, as stated below.

Lemma 1. *If $(\![N]\!) \rightsquigarrow C_1$ and $(\![N]\!) \rightsquigarrow C_2$, then $C_1 \equiv C_2$.*

There is one important design option to consider: what to do with actions that cannot be matched, i.e., processes that will deadlock. There are two alternatives:

$$\frac{}{\mathsf{p} \triangleright \mathsf{q}!\langle e\rangle; B_1 \;\mid\; \mathsf{q} \triangleright \mathsf{p}?; B_2 \xrightarrow{\mathsf{p}.e\,\to\,\mathsf{q}} \mathsf{p} \triangleright B_1 \;\mid\; \mathsf{q} \triangleright B_2} \lfloor\text{S}|\text{Com}\rceil$$

$$\frac{}{\mathsf{p} \triangleright \mathsf{if}\ * \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2 \;\mid\; \mathsf{q} \triangleright \mathsf{p}!\langle e\rangle; B' \xrightarrow{\mathsf{p} \overset{\leftarrow}{=} \mathsf{q}:\mathsf{then}} \mathsf{p} \triangleright B_1 \;\mid\; \mathsf{q} \triangleright B'} \lfloor\text{S}|\text{Then}\rceil$$

Fig. 5. Stateful Processes, Abstract Semantics (selected rules).

restrict extraction to lock-free networks (networks where all processes eventually progress, in the sense of [8]); or extract stuck processes to a new choreography term **1**, with the same semantics as **0**. We choose the latter option for debugging reasons. Specifically, practical applications of extraction may annotate **1** with the code of the deadlocked processes, giving the programmer a chance to see exactly where the system is unsafe, and attempt at fixing it manually. Better yet: since the code to unlock deadlocked processes in process calculi can be efficiently synthesized [8], our method may be integrated with the technique in [8] to suggest an automatic system repair.

Remark 1. If $(\![N]\!) \rightsquigarrow C$ and C does not contain **1**, then N is lock-free. However, even if C contains **1**, N may still be lock-free: the code causing the deadlock may be dead code in a conditional branch that is never chosen during execution.

Extraction is sound: it yields a choreography that is bisimilar to the original network. Also, for finite SP, it behaves as an inverse of EPP.

Theorem 2. *Let N be in finite SP. If $(\![N]\!) \rightsquigarrow C$, then $C \sim N$. Furthermore, if $N = [\![C']\!]$ for some C', then $(\![N]\!) \rightsquigarrow C'$.*

As we show later, the second part of this theorem does not hold in the presence of recursive definitions.

We now restate extraction in terms of a particular graph, which is the hallmark of our development: when we add recursion to SP, we can no longer define extraction as a set of rewriting rules. We first introduce a new abstract semantics for networks, $N \xrightarrow{\alpha} N'$, defined as in Figure 4 except for the rules for value communication and conditionals, which are replaced by those in Figure 5 (we omit the obvious rule $\lfloor$S$|$Else$\rceil$). In particular, conditionals are now nondeterministic. Labels α are like λ but may now contain expressions (see the new rule $\lfloor$S$|$Com$\rceil$); in all other rules, λ is replaced by α. We write $N \xrightarrow{\tilde{\alpha}}{}^{*} N'$ for $N \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} N'$.

Definition 2. *Given a network N, the* Abstract Execution Space (AES) *of N is the directed graph obtained by considering all possible abstract reduction paths from N. Its vertices are all the networks N' such that $N \xrightarrow{\tilde{\alpha}}{}^{*} N'$, and there is an edge between two vertices N_1 and N_2 labeled α if $N_1 \xrightarrow{\alpha} N_2$.*

A Symbolic Execution Graph (SEG) *for N is a subgraph of its AES that contains N and such that each vertex $N' \not\preceq \mathbf{0}$ has either one outgoing edge labeled by an η or two outgoing edges labeled $\mathsf{p} \overset{\leftarrow}{=} \mathsf{q} : \mathsf{then}$ and $\mathsf{p} \overset{\leftarrow}{=} \mathsf{q} : \mathsf{else}$.*

Intuitively, the AES of N represents all possible evolutions of N (each evolution is a path in this graph). A SEG fixes the order of execution of actions, but still abstracts from the state (and thus considers both branches of conditionals). For networks in finite SP, these graphs are finite.

Given a network N, there is a one-to-one correspondence between SEGs for N and choreographies C such that $(\![N]\!) \rightsquigarrow C$. Indeed, given a SEG we can extract a choreography as follows. We start from the initial vertex, labeled N. If there is an outgoing edge with label η to N', we add η to the choreography and continue from N'. If there are two outgoing edges with labels $\mathsf{p} \overset{\leftarrow}{=} \mathsf{q} : \mathsf{then}$ and $\mathsf{p} \overset{\leftarrow}{=} \mathsf{q} : \mathsf{else}$ to N_1 and N_2, respectively, we extract a conditional whose branches are the choreographies extracted by continuing exploration from N_1 and N_2, respectively. When we reach a leaf, we extract $\mathbf{0}$ or $\mathbf{1}$, according to whether its label is equivalent to $\mathbf{0}$ or not. Conversely, we can build a SEG from a particular rewriting of $(\![N]\!)$ by following the choreography actions one at a time.

Treating recursive definitions. We now extend extraction to networks with recursive definitions, using SEGs. We need to be careful with the definition of the AES, since including all possible (abstract) executions now may make it infinite (due to recursion unfolding), and thus extraction may not terminate. To avoid this, we only allow recursive definitions to be unfolded (once) if they occur at the head of a process involved in a reduction. With this restriction, we can define the AES and SEGs for a network as in the finite case. These graphs may now contain cycles: a network may evolve into the same term after a few reductions.

Example 3. Consider the following network.

$$\begin{aligned}&\mathsf{p} \triangleright \mathsf{def}\, X = \mathsf{q}!\langle * \rangle; \mathsf{q}\&\{\textsc{l} : \mathsf{q}!\langle * \rangle; X, \textsc{r} : \mathbf{0}\} \,\mathsf{in}\, \mathsf{q}!\langle * \rangle; X \\ \mid\ & \mathsf{q} \triangleright \mathsf{def}\, Y = \mathsf{p}?; \mathsf{p}?; \mathsf{if}\, * \overset{\leftarrow}{=} \mathsf{r}\, \mathsf{then}\, \mathsf{p} \oplus \textsc{l}; Y\, \mathsf{else}\, \mathsf{p} \oplus \textsc{r}; \mathbf{0}\, \mathsf{in}\, Y \mid \mathsf{r} \triangleright \mathsf{def}\, Z = \mathsf{q}!\langle * \rangle; Z \,\mathsf{in}\, Z\end{aligned}$$

This network generates the AES in Figure 6, which is also its SEG. □

The key insight is that the definitions of recursive procedures are extracted from the loops in the SEG, rather than from the recursive definitions in the source network. This construction typically yields mutually recursive definitions, motivating a small change to CC that does not add expressivity: we replace the constructor $\mathsf{def}\, X = C_2 \,\mathsf{in}\, C_1$ by top-level procedure definitions, in the style of [12]. A choreography now becomes a pair $\langle \mathcal{D}, C \rangle$, where $\mathcal{D} = \{X_i = C_i\}$ and all procedure calls in either C or the C_i are to some X_i defined in D.

Definition 3. *The choreography extracted from a SEG is defined as follows. We annotate each node that has more than one incoming edge with a unique procedure identifier. Then, for every node annotated with an identifier, say X, we replace each of its incoming edges with an edge to a new leaf node that contains a special term X (so now the node annotated with X has no incoming edges). This eliminates all loops in the SEG, allowing us to reuse the extraction procedure for the non-recursive case to extract the desired pair $\langle \mathcal{D}, C \rangle$. We get C by extraction*

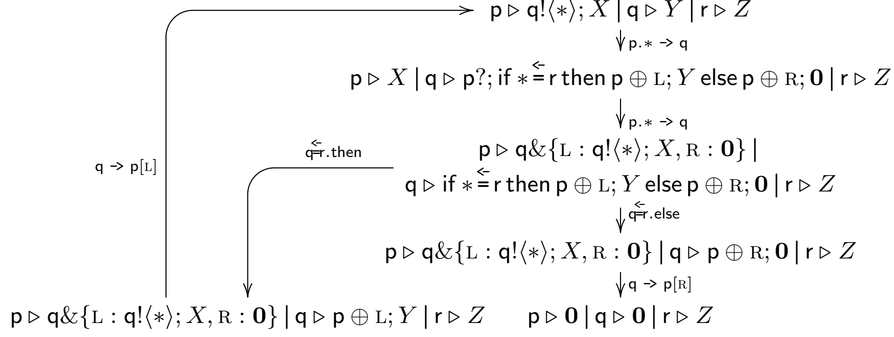


Fig. 6. The AES and SEG for the network in Example 3.

starting from the initial network. Then, for each node that we annotated with an X, we extract a choreographic procedure X in $\mathcal{D}$ that has as body the choreography extracted from the graph that starts from that annotated node. Any new leaf node containing a special term X is extracted as a procedure call X.

Example 4. Consider the SEG in Figure 3. To extract a choreography, we annotate the topmost node with a procedure identifier X and replace the incoming edge to that node with an edge to a new leaf X. We thus extract X to be

$$\mathsf{p}.* \texttt{->} \mathsf{q};\ \mathsf{p}.* \texttt{->} \mathsf{q};\ \mathsf{if}\ \mathsf{q} \stackrel{\leftarrow}{=} \mathsf{r}\ \mathsf{then}\ \mathsf{q} \texttt{->} \mathsf{p}[\mathrm{L}]; X\ \mathsf{else}\ \mathsf{q} \texttt{->} \mathsf{p}[\mathrm{R}]; \mathbf{1}$$

and the extracted choreography itself is simply X. The body of X is not projectable (the branches for r are not mergeable, cf. [11]), but it faithfully describes the behavior of the original network. □

The procedure in Definition 3 always terminates, but sometimes it yields choreographies that starve some processes. As an example, the network

$$\begin{array}{c} \mathsf{p} \triangleright \mathsf{def}\ X = \mathsf{q}!\langle * \rangle; X\ \mathsf{in}\ X \ \mid\ \mathsf{q} \triangleright \mathsf{def}\ Y = \mathsf{p}?; Y\ \mathsf{in}\ Y \\ \mid\ \mathsf{r} \triangleright \mathsf{def}\ Z = \mathsf{s}!\langle * \rangle; Z\ \mathsf{in}\ Z \ \mid\ \mathsf{s} \triangleright \mathsf{def}\ W = \mathsf{r}?; W\ \mathsf{in}\ W \end{array} \qquad (1)$$

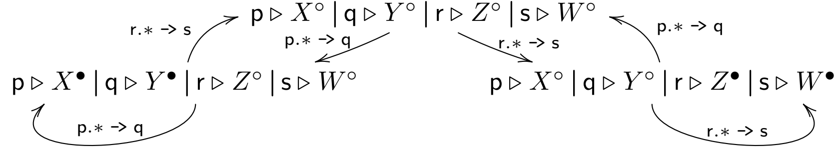
has two SEGs, which extract to the choreographies $\mathsf{def}\ X = \mathsf{p}.* \texttt{->} \mathsf{q}; X\ \mathsf{in}\ X$ and $\mathsf{def}\ X = \mathsf{r}.* \texttt{->} \mathsf{s}; X\ \mathsf{in}\ X$, none of which captures all the behaviors of N.

To avoid this problem, we change the definitions of AES and SEGs slightly. We annotate all procedure calls in networks with either ∘ or •. The node in the AES corresponding to the initial network has all procedure calls annotated with ∘. There is an edge from N to N' with label α if $N \xrightarrow{\alpha} N'$ and the procedure calls in N' are annotated as follows.

- If executing α does not require unfolding procedure calls, then all calls in N' are annotated as in N.
- If executing α requires unfolding procedure calls, then we annotate all the calls in N' introduced by these unfoldings with •. If N' now has *all* procedure calls annotated with •, we change all annotations to ∘.

We then require loops in a SEG to contain a node where every procedure call is annotated with $\circ$. This ensures that every procedure call is unfolded at least once before returning to the same node. This holds even if $\mathsf{p} \triangleright X$ unfolds to a behavior that calls different procedures, but not X: in order to return to the same node, the newly invoked procedures themselves need to be unfolded.

Example 5. The annotated AES for the network (1) is:



This AES now has the following two SEGs:

$$\mathsf{p} \triangleright X^\circ \,|\, \mathsf{q} \triangleright Y^\circ \,|\, \mathsf{r} \triangleright Z^\circ \,|\, \mathsf{s} \triangleright W^\circ \qquad\qquad \mathsf{p} \triangleright X^\circ \,|\, \mathsf{q} \triangleright Y^\circ \,|\, \mathsf{r} \triangleright Z^\circ \,|\, \mathsf{s} \triangleright W^\circ$$

$$\mathsf{r}.* \;\text{->}\; \mathsf{s} \uparrow \;\; \downarrow \mathsf{p}.* \;\text{->}\; \mathsf{q} \qquad\qquad\qquad \mathsf{p}.* \;\text{->}\; \mathsf{q} \uparrow \;\; \downarrow \mathsf{r}.* \;\text{->}\; \mathsf{s}$$

$$\mathsf{p} \triangleright X^\bullet \,|\, \mathsf{q} \triangleright Y^\bullet \,|\, \mathsf{r} \triangleright Z^\circ \,|\, \mathsf{s} \triangleright W^\circ \qquad\qquad \mathsf{p} \triangleright X^\circ \,|\, \mathsf{q} \triangleright Y^\circ \,|\, \mathsf{r} \triangleright Z^\bullet \,|\, \mathsf{s} \triangleright W^\bullet$$

Observe that the self-loops are discarded because they do not go through a node with all $\circ$ annotations. From these SEGs, we can extract two definitions for X:

$$\mathsf{def}\ X = \mathsf{p}.* \;\text{->}\; \mathsf{q}; \mathsf{r}.* \;\text{->}\; \mathsf{s}; X \ \mathsf{in}\ X \qquad \text{and} \qquad \mathsf{def}\ X = \mathsf{r}.* \;\text{->}\; \mathsf{s}; \mathsf{p}.* \;\text{->}\; \mathsf{q}; X \ \mathsf{in}\ X$$

Both of these definitions correctly capture all behaviors of the network. □

A similar situation may occur if there are processes with finite behavior (no procedure calls): the network

$$\mathsf{p} \triangleright \mathsf{def}\ X = \mathsf{q}!\langle * \rangle; X \ \mathsf{in}\ X \;\;|\;\; \mathsf{q} \triangleright \mathsf{def}\ Y = \mathsf{p}?; Y \ \mathsf{in}\ Y \;\;|\;\; \mathsf{r} \triangleright \mathsf{s}!\langle * \rangle \;\;|\;\; \mathsf{s} \triangleright \mathsf{r}?$$

can be extracted to the choreography X, with $X = \mathsf{p}.* \;\text{->}\; \mathsf{q}; X$, where r and s never communicate. Hence, we require that if a node in a SEG has more than one incoming edge (it is a "loop" node) and contains processes with finite behavior, then these processes must be deadlocked (being finite, this is trivially verifiable). This ensures that if finite processes are able to reduce, they cannot be in a loop.

Definition 4. *A SEG for a network N is valid if all its loops:*

- *pass through a node where all recursive calls are marked with $\circ$;*
- *start in a node where all processes with finite behavior are deadlocked.*

A network N extracts to a choreography C if C can be constructed (as in Definition 3) from a valid SEG for N.

Validity implies, however, that there are some non-deadlocked networks that are not extractable, such as

$$\mathsf{p} \triangleright \mathsf{def}\ X = \mathsf{q}!\langle * \rangle; X \ \mathsf{in}\ X \,|\, \mathsf{q} \triangleright \mathsf{def}\ Y = \mathsf{p}?; Y \ \mathsf{in}\ Y \,|\, \mathsf{r} \triangleright \mathsf{def}\ Z = \mathsf{p}?; Z \ \mathsf{in}\ Z$$

for which there is no valid SEG. This is to be expected, since deadlock-freedom is undecidable in SP. We can generalize this observation as a necessary condition for extraction to be defined, in the following theorem.

Theorem 3. *If the AES for a network N does not contain nodes from which a process is always deadlocked, then N is extractable.*

Lemma 1 and the first part of Theorem 2 still hold for extraction in SP with recursion, but the second part of Theorem 2 does not: in general, the projection of a choreography is extracted to a choreography with different procedures, since extraction ignores the actual definitions in the source network.

Theorem 4. *If C is a choreography extracted from a network N, then $N \sim C$.*

We conclude this section with some complexity theoretical considerations.

Lemma 2. *The annotated AES for a network of size n has at most $e^{\frac{2n}{e}}$ vertices.*

Theorem 5. *Extraction from a network of size n terminates in time $O(ne^{\frac{2n}{e}})$.*

As discussed earlier, this time complexity is a dramatic improvement over earlier, comparable work. However, in practice, we may be able to perform even better. Algorithmically, all the required work stems from traversals of the AES, so any reduction in its (explored) size will lead to proportional runtime improvements. Thus, instead of first computing the entire AES and then a valid SEG, we can compute the relevant parts of the AES lazily as we need them, so parts of the AES that are never explored while computing a valid SEG are never generated.

5 Asynchrony

We now discuss an asynchronous semantics for SP, with which we can express new safe behaviors. Most notably, SP can now express asynchronous exchange (Example 6). We also show a novel choreography primitive that successfully captures this pattern, which cannot be described in previous works on choreographic programming, and extend our algorithm to extract it from networks.

Asynchronous SP. Asynchronous communication can be added to SP using standard techniques for process calculi. In the semantics of networks, we add a FIFO queue for each pair of processes. Communications now synchronize with these queues: send actions append a message in the queue of the receiver, and receive actions remove the first message from the queue of the receiver (see [12] for a formalization in an extension of SP).

Example 6. The network $\mathsf{p} \triangleright \mathsf{q}!\langle * \rangle; \mathsf{q}? \mid \mathsf{q} \triangleright \mathsf{p}!\langle * \rangle; \mathsf{p}?$ exemplifies the pattern of asynchronous exchange. This network is deadlocked in synchronous SP, but runs without errors in asynchronous SP: both p and q can send their respective values, becoming ready to receive each other's messages. This behavior is not representable in any previous work on choreographies (including CC from § 3), since all choreographies presented so far can only describe processes that are not deadlocked under a synchronous semantics (see [12] for a formal argument). □

The multicom. The situation in Example 6 is prototypical of programs that are safe only in an asynchronous setting: a group of processes wants to send messages to a group of receivers, with circular dependencies among communications.

We deal with this situation by means of a new choreography action, which we call a *multicom*. Syntactically, a multicom is a list of communication actions with distinct receivers, which we write $(\tilde{\eta})$. In the unary case, we obtain the usual communications and selections; by removing these from the syntax of CC and adding the multicom, we obtain a more expressive calculus with fewer primitives. The semantics of multicom is given by the following rule, which generalizes (and replaces) both $\lfloor$C|Com$\rceil$ and $\lfloor$C|Sel$\rceil$.

$$\frac{I = \{i \mid \mathsf{p}_i.e_i \texttt{ -> } \mathsf{q}_i \in \tilde{\eta}\} \qquad v_i = e_i[\sigma(\mathsf{p}_i)/*]}{(\tilde{\eta}); C, \sigma \xrightarrow{(\tilde{\eta})[e_i/v_i]_{i \in I}} C, \sigma[\mathsf{q}_i \mapsto v_i]_{i \in I}} \lfloor\text{C|MCom}\rceil$$

Structural precongruence rules for the multicom are motivated by its intuitive semantics: actions inside a multicom can be permuted as long as the senders differ, and sequential multicoms can be merged as long as they do not share receivers and there are no sequential constraints between them (i.e., none of the receivers in the first multicom is a sender in the second one).

$$\frac{\mathsf{pn}(\eta_1) \cap \mathsf{pn}(\eta_2) = \emptyset}{(\ldots, \eta_1, \eta_2, \ldots) \equiv (\ldots, \eta_2, \eta_1, \ldots)} \lfloor\text{C|MCom-Perm}\rceil$$

$$\frac{\mathsf{rcv}(\eta) \cap \mathsf{rcv}(\nu) = \emptyset \qquad \mathsf{rcv}(\tilde{\eta}) \cap \mathsf{snd}(\tilde{\nu}) = \emptyset}{(\tilde{\eta}) ; (\tilde{\nu}) \equiv (\tilde{\eta}, \tilde{\nu})} \lfloor\text{C|MCom-MCom}\rceil$$

From these rules we can derive all instances of $\lfloor$C|Eta-Eta$\rceil$, e.g.:

$$\mathsf{p}.* \texttt{ -> } \mathsf{q}; \mathsf{r}.* \texttt{ -> } \mathsf{s} \equiv \begin{pmatrix} \mathsf{p}.* \texttt{ -> } \mathsf{q} \\ \mathsf{r}.* \texttt{ -> } \mathsf{s} \end{pmatrix} \equiv \begin{pmatrix} \mathsf{r}.* \texttt{ -> } \mathsf{s} \\ \mathsf{p}.* \texttt{ -> } \mathsf{q} \end{pmatrix} \equiv \mathsf{r}.* \texttt{ -> } \mathsf{s}; \mathsf{p}.* \texttt{ -> } \mathsf{q}$$

The problematic program in Example 6 can now be written as $\begin{pmatrix} \mathsf{p}.* \texttt{ -> } \mathsf{q} \\ \mathsf{q}.* \texttt{ -> } \mathsf{p} \end{pmatrix}$.

Structural precongruence rules for multicom also allow us to define a normal form for choreographies, where no multicom can be split in smaller multicoms.

Extraction. In order to extract choreographies containing multicoms, we alter the definition of the AES for a process network by allowing multicoms as labels for the edges. These can be computed using the following iterative algorithm.

1. For a process p with behavior $\mathsf{q}!\langle e\rangle; B$ (or $\mathsf{q} \oplus l; B$), set $\mathsf{actions} = \emptyset$ and $\mathsf{waiting} = \{\mathsf{p}.e \texttt{ -> } \mathsf{q}\}$ (resp. $\mathsf{waiting} = \{\mathsf{p} \texttt{ -> } \mathsf{q}[l]\}$).
2. While $\mathsf{waiting} \neq \emptyset$:
   (a) Move an action η from $\mathsf{waiting}$ to $\mathsf{actions}$. Assume η is of the form $\mathsf{r}.e \texttt{ -> } \mathsf{s}$ (the case for label selection is similar).
   (b) If the behavior of s is of the form $a_1; \ldots; a_k; \mathsf{r}?; B$ where each a_i is either the sending of a value or a label selection, then: for each a_i, if the corresponding choreography action is not in $\mathsf{actions}$, add it to $\mathsf{waiting}$.

3. Return $\mathsf{actions}$.

This algorithm may fail (the behavior of s in step 2(b) is not of the required form), in which case the action initially chosen cannot be unblocked by a multicom.

Example 7. Consider the network from Example 6. Starting with action $\mathsf{q}!\langle * \rangle$ at process p, we initialize $\mathsf{actions} = \emptyset$ and $\mathsf{waiting} = \{\mathsf{p}.* \text{ -> } \mathsf{q}\}$. We pick the action $\mathsf{p}.* \text{ -> } \mathsf{q}$ from $\mathsf{waiting}$ and move it to $\mathsf{actions}$. The behavior of q is $\mathsf{p}!\langle * \rangle; \mathsf{p}?$, which is of the form described in step 2(b); the choreography action corresponding to $\mathsf{p}!\langle * \rangle$ is $\mathsf{q}.* \text{ -> } \mathsf{p}$, so we add this action to $\mathsf{waiting}$, obtaining $\mathsf{actions} = \{\mathsf{p}.* \text{ -> } \mathsf{q}\}$ and $\mathsf{waiting} = \{\mathsf{q}.* \text{ -> } \mathsf{p}\}$. Now we consider the action $\mathsf{q}.* \text{ -> } \mathsf{p}$, which we move from $\mathsf{waiting}$ to action, and look at p's behavior, which is $\mathsf{q}!\langle * \rangle; \mathsf{q}?$. The choreography action corresponding to $\mathsf{q}!\langle * \rangle$ is $\mathsf{p}.* \text{ -> } \mathsf{q}$, which is already in $\mathsf{actions}$, so we do not change $\mathsf{waiting}$. The set $\mathsf{waiting}$ is now empty, and the algorithm terminates, returning $\begin{pmatrix} \mathsf{p}.* \text{ -> } \mathsf{q} \\ \mathsf{q}.* \text{ -> } \mathsf{p} \end{pmatrix}$. We would obtain the equivalent $\begin{pmatrix} \mathsf{q}.* \text{ -> } \mathsf{p} \\ \mathsf{p}.* \text{ -> } \mathsf{q} \end{pmatrix}$ by starting with the send action at q. □

Example 8. As a more sophisticated example, we show how our new choreographies with multicom can model the alternating 2-bit protocol. Here, Alice alternates between sending a 0 and a 1 to Bob; in turn, Bob sends an acknowledgment for every bit he receives, and Alice waits for the acknowledgment before sending another copy of the same bit. Since we are in an asynchronous semantics, we only consider the time when the messages arrive. With this in mind, we can write this protocol as the following network.

$$\begin{aligned} &\mathsf{a} \triangleright \mathsf{def}\ X = (\mathsf{b}?; \mathsf{b}!\langle 0 \rangle; \mathsf{b}?; \mathsf{b}!\langle 1 \rangle; X)\ \mathsf{in}\ (\mathsf{b}!\langle 0 \rangle; \mathsf{b}!\langle 1 \rangle; X) \\ |\ \ &\mathsf{b} \triangleright \mathsf{def}\ Y = (\mathsf{a}?; \mathsf{a}!\langle \mathsf{ack}_0 \rangle; \mathsf{a}?; \mathsf{a}!\langle \mathsf{ack}_1 \rangle; Y)\ \mathsf{in}\ Y \end{aligned}$$

This implementation imposes exactly the dependencies dictated by the protocol. For example, Alice can receive Bob's acknowledgment to the first 0 before or after Bob receives the first 1. This network extracts to the choreography

$$\mathsf{a}.0 \text{ -> } \mathsf{b}; X \qquad \text{where} \qquad X = \begin{pmatrix} \mathsf{a}.1 \text{ -> } \mathsf{b} \\ \mathsf{b}.\mathsf{ack}_0 \text{ -> } \mathsf{a} \end{pmatrix}; \begin{pmatrix} \mathsf{a}.0 \text{ -> } \mathsf{b} \\ \mathsf{b}.\mathsf{ack}_1 \text{ -> } \mathsf{a} \end{pmatrix}; X$$

which is a simple and elegant representation of the alternating 2-bit protocol. □

Extraction for asynchronous SP is still sound, but behavioral equivalence is now an expansion [1, 24], as each communication now takes two steps in asynchronous SP. Its complexity is also no larger than for the synchronous case. The algorithm computing the multicom takes linear time in the size of the multicom produced. Via a one-time preprocessing of the network, we can assume direct references from communication terms in one process to the process it directs its communication at, and from there to the current state of that process. Other than the above, all constant steps in the algorithm can be seen as an extension of the multicom. Since adding a communication to a multicom removes a potential node in the AES (as we are combining communications), the worst-case time

complexity is no worse than in the synchronous case. In practice, this complexity actually gets better when larger multicoms are created, since building these is a much cheaper local operation than exploring graphs that would be larger in terms of nodes as well as edges without the multicoms.

6 Extensions and Applications

We discuss some straightforward modifications of our extraction to cover other scenarios occurring in the literature.

More expressive communications and processes. In real-world contexts, the values stored and communicated by processes are typed, and the receiver process can also specify how to treat incoming messages [12]. This means that communication actions now have the form $\mathsf{p}.e$ -> $\mathsf{q}.f$, where f is the function consuming the received message, and systems may deadlock because of typing errors. Our construction applies without changes to this scenario.

Some works allow processes to store several values, used via variables [5, 6]. Again, dealing with this situation does not require any changes to our algorithm.

Local conditionals. Many choreography models allow for a local conditional construct, i.e., $\mathsf{if}\ \mathsf{p}.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ [6, 21, 14]. Dealing with this construct is simple: the if and then transitions now can occur whenever a process has a conditional as top action, since they no longer require synchronization with other processes.

Choreography Specifications. So far, we have considered choreographies that describe concrete implementations, i.e., processes are equipped with storage and local computational capabilities. However, choreographies have also been advocated for the specification of communication protocols. Most notably, multiparty session types use choreographies to define types used in the verification of process calculi [17]. While there are multiple variants of multiparty session types, the one so far most used in practice is almost identical to a simplification of SP. In this variant, each pair of participants has a dedicated channel, and communication actions refer directly to the intended sender/recipient as in SP (see, e.g., the theory of [6, 21, 9, 10] and the practical implementations in [16, 23, 20]). To obtain multiparty session types from SP (and CC), we just need to: remove the capability of storing values at processes; replace message values with constants (representing types, which could also be extended to subtyping in the straightforward way); and make conditionals nondeterministic (since in types we abstract from the precise values and expression used by the evaluator). These modifications do not require any significant change to our approach, since our AES already abstracts from data and thus our treatment of the conditional is already nondeterministic. For reference, we can simply treat the standard construct for an internal choice at a process p – $C_1 \oplus_{\mathsf{p}} C_2$ – as syntactic sugar for a local conditional like $\mathsf{if}\ \mathsf{p}.\mathsf{coinflip}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$.

References

1. S. Arun-Kumar and Matthew Hennessy. An efficiency preorder for processes. *Acta Inf.*, 29(8):737–760, 1992.
2. Samik Basu and Tevfik Bultan. Choreography conformance via synchronizability. In *WWW*, pages 795–804, 2011.
3. Business Process Model and Notation. `http://www.omg.org/spec/BPMN/2.0/`.
4. D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, April 1983.
5. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.
6. M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In R. Giacobazzi and R. Cousot, editors, *POPL*, pages 263–274. ACM, 2013.
7. M. Carbone, F. Montesi, and C. Schürmann. Choreographies, logically. In P. Baldan and D. Gorla, editors, *CONCUR*, volume 8704 of *LNCS*, pages 47–62. Springer, 2014.
8. Marco Carbone, Ornela Dardha, and Fabrizio Montesi. Progress as compositional lock-freedom. In *Proc. of COORDINATION*, pages 49–64, 2014.
9. Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, volume 59 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
10. Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):238–302, 2016.
11. L. Cruz-Filipe and F. Montesi. A Core Model for Choreographic Programming. In *FACS*, LNCS. Springer, accepted for publication.
12. Luís Cruz-Filipe and Fabrizio Montesi. Choreographies, divided and conquered. *CoRR*, abs/1602.03729, 2016. Submitted for publication.
13. Luís Cruz-Filipe and Fabrizio Montesi. Choreographies in practice. In Elvira Albert and Ivan Lanese, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*, pages 114–123. Springer, 2016.
14. M. Dalla Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, and J. Mauro. Dynamic choreographies – safe runtime updates of distributed applications. In T. Holvoet and M. Viroli, editors, *COORDINATION*, volume 9037 of *LNCS*, pages 67–82. Springer, 2015.
15. Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In Helmut Seidl, editor, *ESOP*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.
16. K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, and N. Yoshida. Scribbling interactions with a formal foundation. In R. Natarajan and A.K. Ojo, editors, *ICDCIT*, volume 6536 of *LNCS*, pages 55–75. Springer, 2011.
17. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9, 2016.

18. Julien Lange and Emilio Tuosto. Synthesising choreographies from local session types. In *CONCUR*, pages 225–239, 2012.
19. Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 221–232. ACM, 2015.
20. F. Montesi. *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen, 2013. http://fabriziomontesi.com/files/choreographic_programming.pdf.
21. F. Montesi and N. Yoshida. Compositional choreographies. In P.R. D'Argenio and H.C. Melgratti, editors, *CONCUR*, volume 8052 of *LNCS*, pages 425–439. Springer, 2013.
22. Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP*, pages 316–332, 2009.
23. Nicholas Ng and Nobuko Yoshida. Pabble: Parameterised scribble for parallel programming. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy, February 12-14, 2014*, pages 707–714. IEEE Computer Society, 2014.
24. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
25. W3C WS-CDL Working Group. Web services choreography description language version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/, 2004.

Full semantics of SP

The full semantics of SP is given in Figure 7.

EndPoint Projection

EPP is inductively defined by the rules in Figure 8. EPP produces a parallel composition with one process for each one process name in the original choreography. The rules for projecting process behaviors follow the intuition of projecting the local action performed by the process of interest. The rules for projecting recursive definitions and calls assume that procedure names have been annotated with the process names appearing inside the body of the procedure, in order to avoid projecting unnecessary procedure code (see [11]). The rule for projecting a conditional uses the partial merging operator $\sqcup$ to merge the possible behaviors of a process that does not know which branch will be chosen. Merging is a homomorphic binary operator; for all terms but branchings it requires isomorphism, e.g.: $\mathsf{q}!\langle e\rangle; B \sqcup \mathsf{q}!\langle e\rangle; B' = \mathsf{q}!\langle e\rangle; (B \sqcup B')$. Branching terms can have unmergeable continuations, as long as they are guarded by distinct labels. In this case, merge returns a larger branching including all options (merging branches with the same label):

$$\mathsf{p}\&\{l_i : B_i\}_{i\in J} \sqcup \mathsf{p}\&\{l_i : B'_i\}_{i\in K} = \\ \mathsf{p}\&\big(\{l_i : (B_i \sqcup B'_i)\}_{i\in J\cap K} \cup \{l_i : B_i\}_{i\in J\setminus K} \cup \{l_i : B'_i\}_{i\in K\setminus J}\big)$$

$$\frac{e[\sigma(\mathsf{p})/*] \downarrow v}{\mathsf{p} \triangleright \mathsf{q}!\langle e \rangle; B_1 \;\mid\; \mathsf{q} \triangleright \mathsf{p}?; B_2,\ \sigma \xrightarrow{\mathsf{p}.v\ \texttt{->}\ \mathsf{q}} \mathsf{p} \triangleright B_1 \;\mid\; \mathsf{q} \triangleright B_2,\ \sigma[\mathsf{q} \mapsto v]} \lfloor\text{S|Com}\rceil$$

$$\frac{\mathsf{p} \triangleright B_1 \;\mid\; N,\ \sigma \xrightarrow{\lambda} \mathsf{p} \triangleright B_1' \;\mid\; N',\ \sigma'}{\mathsf{p} \triangleright \mathsf{def}\ X = B_2\ \mathsf{in}\ B_1 \;\mid\; N,\ \sigma \xrightarrow{\lambda} \mathsf{p} \triangleright \mathsf{def}\ X = B_2\ \mathsf{in}\ B_1' \;\mid\; N',\ \sigma'} \lfloor\text{S|Ctx}\rceil$$

$$\frac{j \in I}{\mathsf{p} \triangleright \mathsf{q} \oplus l_j; B \;\mid\; \mathsf{q} \triangleright \mathsf{p}\&\{l_i : B_i\}_{i \in I},\ \sigma \xrightarrow{\mathsf{p}\ \texttt{->}\ \mathsf{q}[l]} \mathsf{p} \triangleright B \;\mid\; \mathsf{q} \triangleright B_j,\ \sigma} \lfloor\text{S|Sel}\rceil$$

$$\frac{e[\sigma(\mathsf{q})/*] \downarrow \sigma(\mathsf{p})}{\mathsf{p} \triangleright \mathsf{if}\ * \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2 \;\mid\; \mathsf{q} \triangleright \mathsf{p}!\langle e \rangle; B',\ \sigma \xrightarrow{\mathsf{p} \overset{\leftarrow}{=} \mathsf{q}:\mathsf{then}} \mathsf{p} \triangleright B_1 \;\mid\; \mathsf{q} \triangleright B',\ \sigma} \lfloor\text{S|Then}\rceil$$

$$\frac{e[\sigma(\mathsf{q})/*] \not\downarrow \sigma(\mathsf{p})}{\mathsf{p} \triangleright \mathsf{if}\ * \overset{\leftarrow}{=} \mathsf{q}\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2 \;\mid\; \mathsf{q} \triangleright \mathsf{p}!\langle e \rangle; B',\ \sigma \xrightarrow{\mathsf{p} \overset{\leftarrow}{=} \mathsf{q}:\mathsf{else}} \mathsf{p} \triangleright B_2 \;\mid\; \mathsf{q} \triangleright B',\ \sigma} \lfloor\text{S|Else}\rceil$$

$$\frac{N,\ \sigma \xrightarrow{\lambda} N',\ \sigma'}{N \mid M,\ \sigma \xrightarrow{\lambda} N' \mid M,\ \sigma'} \lfloor\text{S|Par}\rceil \qquad \frac{N \preceq M \quad M, \sigma \xrightarrow{\lambda} M', \sigma' \quad M' \preceq N'}{N, \sigma \xrightarrow{\lambda} N', \sigma'} \lfloor\text{S|Struct}\rceil$$

Fig. 7. Stateful Processes, Semantics.

Proofs of results on extraction (finite case)

Proof (Lemma 1). By definition, $\rightsquigarrow$ has the diamond property, and all the possible diamonds correspond exactly to rules in the definition of the structural precongruence relation for CC. The thesis then follows by induction on the number of rewriting steps in $N \rightsquigarrow^* C_1$.

Proof (Theorem 2). Straightforward by structural induction on C.

Remark 2. The extracted choreography can be exponential in the size of the original network. Consider the family of networks $\mathcal{N}_n$ defined as follows.

$$\mathcal{N}_n = \prod_{i=1}^{n} \left(\mathsf{p}_{2i-1} \triangleright \mathsf{if}\ * \overset{\leftarrow}{=} \mathsf{p}_{2i}\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathbf{0} \mid \mathsf{p}_{2i} \triangleright \mathsf{p}_{2i-1}!\langle e \rangle \right)$$

$\mathcal{N}_n$ contains exactly $2n$ actions, of which half are conditionals and half are message sends. A straightforward induction proof establishes that every choreography C such that $(\![\mathcal{N}_n]\!) \rightsquigarrow^* C$ contains $2^n - 1$ conditionals (and no other actions).

Encoding top-level definitions in CC

We show how to encode top-level definitions in the original syntax of CC. We illustrate the exponential growth by means of a choreography with two mutually recursive definitions: $\langle \{X = C_X, Y = C_Y\}, C \rangle$ where both C_X, C_Y and C contain

$$[\![C,\sigma]\!] = \prod_{\mathsf{p}\in\mathsf{pn}(C)} \mathsf{p} \triangleright_{\sigma(\mathsf{p})} [\![C]\!]_{\mathsf{p}}$$

$$[\![\mathsf{p}.e \text{ -> } \mathsf{q}; C]\!]_{\mathsf{r}} = \begin{cases} \mathsf{q}!\langle e\rangle; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r}=\mathsf{p} \\ \mathsf{p}?; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r}=\mathsf{q} \\ [\![C]\!]_{\mathsf{r}} & \text{o.w.} \end{cases} \qquad [\![\mathsf{p} \text{ -> } \mathsf{q}[l]; C]\!]_{\mathsf{r}} = \begin{cases} \mathsf{q}\oplus l; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r}=\mathsf{p} \\ \mathsf{p}\&\{l : [\![C]\!]_{\mathsf{r}}\} & \text{if } \mathsf{r}=\mathsf{q} \\ [\![C]\!]_{\mathsf{r}} & \text{o.w.} \end{cases}$$

$$[\![\mathsf{if}\ \mathsf{p} \overset{\Leftarrow}{=} \mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2]\!]_{\mathsf{r}} = \begin{cases} \mathsf{if}\ * \overset{\Leftarrow}{=} \mathsf{q}\ \mathsf{then}\ [\![C_1]\!]_{\mathsf{r}}\ \mathsf{else}\ [\![C_2]\!]_{\mathsf{r}} & \text{if } \mathsf{r}=\mathsf{p} \\ \mathsf{p}!\langle *\rangle; ([\![C_1]\!]_{\mathsf{r}} \sqcup [\![C_2]\!]_{\mathsf{r}}) & \text{if } \mathsf{r}=\mathsf{q} \\ [\![C_1]\!]_{\mathsf{r}} \sqcup [\![C_2]\!]_{\mathsf{r}} & \text{o.w.} \end{cases} \qquad [\![\mathbf{0}]\!]_{\mathsf{r}} = \mathbf{0}$$

$$[\![\mathsf{def}\ X^{\tilde{\mathsf{p}}} = C_2\ \mathsf{in}\ C_1]\!]_{\mathsf{r}} = \begin{cases} \mathsf{def}\ X = [\![C_2]\!]_{\mathsf{r}}\ \mathsf{in}\ [\![C_1]\!]_{\mathsf{r}} & \text{if } \mathsf{r}\in\tilde{\mathsf{p}} \\ [\![C_1]\!]_{\mathsf{r}} & \text{o.w.} \end{cases} \qquad [\![X^{\tilde{\mathsf{p}}}]\!]_{\mathsf{r}} = \begin{cases} X & \text{if } \mathsf{r}\in\tilde{\mathsf{p}} \\ \mathbf{0} & \text{o.w.} \end{cases}$$

Fig. 8. Core Choreographies, EndPoint Projection (EPP).

calls to X and Y. If we try to define it as a choreography of the form $\mathsf{def}\ X = C_2\ \mathsf{in}\ C_1$, then both C_1 and C_2 must be able to invoke Y, which means we have to duplicate the definition of Y, obtaining

$$\mathsf{def}\ X = (\mathsf{def}\ Y = C_Y\ \mathsf{in}\ C_X)\ \mathsf{in}\ (\mathsf{def}\ Y' = C_Y[Y'/Y]\ \mathsf{in}\ C[Y'/Y])$$

and the terms in parentheses correspond to the choreographies $\langle\{Y = C_Y\}, C_X\rangle$ and $\langle\{Y = C_Y\}, C\rangle$ (where they are allowed to use X).

In general, we can therefore encode procedure definitions at the top by means of an operator $\{\![\mathcal{D}, C]\!\}$ defined as

$$\{\![\emptyset, C]\!\} = C$$
$$\{\![\{X = C_X\} \cup \mathcal{D}, C\}]\!\} = \mathsf{def}\ X = \{\![\mathcal{D}, C_X]\!\}\ \mathsf{in}\ \{\![\mathcal{D}, C]\!\}$$

where we rely on α-renaming to obtain different names for the procedures defined in the two recursive calls to $\{\![\cdot]\!\}$.

Theorem 6. *The choreographies $\langle\mathcal{D}, C\rangle$ and $\{\![\mathcal{D}, C]\!\}$ are behaviorally equivalent.*

Proofs of results on extraction (general case)

Proof (Theorem 3). We describe an algorithm to find a valid SEG in the AES, if it exists. To make it clearer, we first describe how one (possibly invalid) SEG could be found. Start with the node representing the initial network and move along edges in the graph, noting that if an edge labeled $\xrightarrow{\mathsf{p} \overset{\Leftarrow}{=} \mathsf{q}:\mathsf{then}}$ is chosen, then the path containing the corresponding else action must also be taken (and reciprocally). We keep exploring the graph until all paths explored end or loop

into nodes already explored. Given the method of construction, we can view this as a tree with loops back to earlier nodes. This construction will give us one SEG.

To find a *valid* SEG, we backtrack over all the nodes where there were alternative path continuations. We enumerate nodes consecutively as we explore them, so if their number is different from the initial dummy value, we have found a loop. From such a node where we just discovered that a loop started, we can traverse the loop again by going towards the node with larger (initialized) number whenever there is a choice, and check for the existence of an all-white node somewhere on the loop. If found, we proceed recursively with the latest unexplored branch. Otherwise, we backtrack, choosing the most recent unexplored choice.

If there are no more possible unexplored alternatives, then one of the processes in the node where the loop started is deadlocked in all subsequent states, contradicting the hypothesis.

Proof (Theorem 4, sketch). Let N be a network and C be a choreography extracted from N. Define a relation $\mathcal{R} \subseteq \mathcal{C} \times \mathcal{N}$, where $\mathcal{C} = \{C' \mid C \to^* C'\}$ and $\mathcal{N} = \{N' \mid N \to^* N'\}$, as follows: $C'\mathcal{R}N'$ if $C \to^* C'$ and $N \to^* N'$ with the same sequence of actions. We prove that $\mathcal{R}$ is a bisimulation by induction on the length of this sequence.

If the sequence is empty and $C \to C''$, then clearly $N \to N''$ with the same action, since C is defined by choosing an action that N can make. If $N \to N''$, there are two cases; the interesting one is when the action taken is not the same as specified at the top of C. Note that the processes involved are not able to participate in any other reductions, so the action remains enabled in all execution paths of N, in particular in that taken by C, and is swappable with every action in C. Then we simply need to show that C eventually takes this action, which is guaranteed by the fairness conditions imposed in Definition 4.

If this sequence is not empty, the proof for the case when C' makes a move remains the same. For the case when N' makes a move, we can make a similar argument by considering the actions from N to N' that occur at the top level in C, which are necessarily swappable with the remaining ones; so C can reduce to a choreography C^*, which can execute the action executed by N' (as in the base case) and then the remaining actions in the reduction from N to N' to C''. Again by swapping, also $C' \to C''$.

Proof (Lemma 2). Let N be a network with p processes of sizes n_1 through n_p, where the size of a process is the number of nodes in an abstract syntax tree representing the syntactical term. We let $n = \sum_{i=1}^{p} n_i$ denote the size of N.

Since recursive definitions are unfolded only when they occur at the top of a behavior, a process of size n_i can give rise to at most n_i different terms when all possible reductions are considered. Thus, N can reduce to at most $\prod_{i=1}^{p} n_i$ different terms. Since the reductions give rise to the edges in the graph, this is also an upper bound on the number of edges, so the graph is sparse. By the AM-GM inequality, $\prod_{i=1}^{p} n_i$ is maximized when all the n_i are equal, where it evaluates

to $\left(\frac{n}{p}\right)^p$. We now consider annotations. We observe that all procedure calls in the same process must be marked with the same token, so there are at most 2^p annotations for each network, giving a total upper bound of $2^p(\frac{n}{p})^p = (\frac{2n}{p})^p$ different nodes in the AES. This expression attains its maximum when $p = \frac{2n}{e}$, where e is Euler's number, giving the upper bound of $e^{\frac{2n}{e}}$ nodes in the AES graph. □

Proof (Theorem 5). Constructing the graph can be done iteratively by maintaining a set of unexplored nodes. Whenever an unexplored node is examined, the possible reductions lead to new terms; by keeping all created nodes in a search structure, we can check if the node already exists (and get a reference to it) in logarithmic time in the number of nodes (which is $O(n)$) and linear time in the size of the term (which is also logarithmic in the size of the graph). Thus, the AES can be constructed with overall complexity $O(ne^{\frac{2n}{e}})$.

To extract a valid SEG from the AES, we perform a graph traversal, which is linear in the size of the graph. There is the one complication, however: checking that loops contain an all-white node. Rechecking the entire loop could potentially move the overall complexity to factorial time; we explain how to handle this without increasing the asymptotic time complexity. Recall from the proof of Theorem 3 that we stop our search and start backtracking when we discover a loop; we thus conceptually have a path from the start node to our current node at all times, and the path behaves in a stack-like manner. We introduce an explicit stack as an auxiliary data structure. Each node on the current path has a pointer to its entry on the stack. An item on the stack contains a boolean stating if its corresponding node is all-white and a counter of how many white nodes can be found further down on the stack. This information can easily be maintained in constant time as we push and pop elements in connection with running the backtracking algorithm. When we encounter a loop, we follow the pointer to the node's associated stack item and check the counter, c. The loop just found has at least one white node if and only if the counter of the top item on the stack is strictly greater than c.