A Graphical Approach to Progress for Structured Communication in Web Services*

Marco Carbone Søren Debois
IT University of Copenhagen
Copenhagen, Denmark
{carbonem,debois}@itu.dk

We investigate a graphical representation of session invocation interdependency in order to prove progress for the π-calculus with sessions under the usual session typing discipline. We show that those processes whose associated dependency graph is acyclic can be brought to reduce. We call such processes *transparent processes*. Additionally, we prove that for well-typed processes where services contain no free names, such acyclicity is preserved by the reduction semantics.

Our results encompass programs (processes containing neither free nor restricted session channels) and higher-order sessions (delegation). Furthermore, we give examples suggesting that transparent processes constitute a large enough class of processes with progress to have applications in modern session-based programming languages for web services.

1 Introduction

Due to fast-growing technologies and exponential growth of the Internet and the world-wide web, computing systems and software based on communication are becoming the norm rather than the exception. In particular, *Web Services (WS)* is today a crucial ingredient in many such systems. The W3C, the world-wide web's governing body, defines WS as *"a software system designed to support interoperable machine-to-machine interaction over a network"* [18]. Abstractly, we think of WS as running processes, each identified by some name, which can be repeatedly invoked by clients or other services. Such an invocation spawns a new thread of the service which handles the actual interaction between service and client. This interaction, a sequence of input-output operations, is often referred to as *session*.

Recently, sessions have been the subject of intense research. Most pertinent to the present paper, calculi for concurrency have been equipped with typing system ensuring session safety [6, 17, 11]. Such session typing systems have in turn given rise to a host of programming languages using session types to control concurrency, e.g., [12, 16, 14] to cite a few. Such languages derive their strengths and practicality directly from our theoretical understanding of the underlying calculi.

One class of theoretical problems with direct ramifications for programming languages is that of ensuring *progress* of sessions, that is, statically ensuring that protocols do not inadvertently *get stuck* or *deadlock* [10, 8, 9, 2]. In the present work, we investigate a graphical representation of session invocation interdependency and exploit its properties for proving progress. As a result, we identify a new class of processes, which we call the *transparent* processes. This class advances the state of the art in two directions: (1) it includes processes not hitherto identified as having progress and (2) it is characterised by a simple syntactic criterion which requires no special-purpose typing system or inference and is computable in linear time with respect to the number of nodes in the abstract syntax tree of a process.

*This work is funded in part by the Danish Research Agency (grant no.: 2106-080046) and the IT University of Copenhagen (the Jingling Genies projects). Authors are listed alphabetically by last name.

ICE 2010
EPTCS ??, 20??, pp. 1–15, doi:10.4204/EPTCS.??.??

We shall argue that the class we consider has specific practical motivation and, in combination with (2), it thus seems to be of potentially direct practical importance.

In session-based systems such as WS, caller and callee play central roles. Traditional calculi take a completely symmetric approach to such roles. We claim that this approach is perhaps not fully aligned with the practicalities of WS: the communicating parties, clients and services, are not equal but rather *inherently asymmetric*. Whereas clients can be anything, services must be mutually independent. Thus, we shall assume that *a service can never depend on previously opened communications*. This assumption leads us to the class of transparent processes, while simultaneously assuring practical relevance.

Formally, we work in a π-calculus with sessions and session types á là [11]. In this model, the above assumption becomes simply that no service contain free session channels. This assumption is by itself sufficient to guarantee progress, without any extra constraints on well-typed processes. Additionally, the progress result of the present paper actually goes beyond such self-contained services i.e. transparent processes are a larger class than just closed services. Technically, we adopt a near-standard session-typing discipline similar to the one used in [6, 7]. That services cannot rely on already open communications is reflected by the following only non-standard typing rule for services. By example:

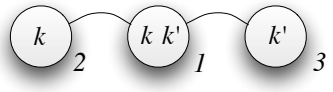
$$(\text{T-SERV}) \quad \frac{\Gamma,\ \mathtt{buy} : \langle\alpha\rangle \quad \vdash \quad \boxed{P} \quad \triangleright \boxed{k : \alpha}}{\Gamma,\ \mathtt{buy} : \langle\alpha\rangle \quad \vdash \quad \boxed{\mathtt{buy}(k).\, P} \quad \triangleright \boxed{\emptyset}}$$

Above, we have framed those points relevant to the discussion. The term $\mathtt{buy}(k).\, P$ denotes a service named buy which, upon invocation, will create some private channel k and use it in its body P for exchanging messages with the invoker. The typing rule says that the body P has access only to the new session k (expressed in the premise as $k : \alpha$ where α describes how k will be used in P), and not to any other previously opened session. Technically, this is just a small restriction to standard session typing [11], thus any process typeable in the present system will be typeable in the standard one. On the other hand, this restriction is practically reasonable [6].

The central idea for proving the progress property is to focus on the development of *session dependency graphs*, first introduced in [7]. These capture the key intuition in our transparent processes. Whereas much work in progress on deadlock works essentially by ordering the sequential use of channels (e.g. [13, 9, 3]), the present work, *qua* the session dependency graph, works instead by ordering the running threads of a system by their pairwise sharing of sessions. Consider the following example; we call prefixes in a parallel composition *threads*:

$$\underbrace{k?(x).k'!\langle x\rangle}_{1} \mid \underbrace{k!\langle 5\rangle}_{2} \mid \underbrace{k'?(y)}_{3}$$

Above, thread 1 receives a value on session channel k and then forwards it to another session k'. Thread 2 sends value 5 on session k while thread 3 waits for a message to be sent on session k'. Each of the three processes above represents a node in the session dependency graph. Moreover, an edge between two nodes is in the graph whenever their corresponding processes share a free session channel. Graphically:



The *transparent* processes are simply those processes which have acyclic session dependency graph at every sub-term. Acyclicity at top-level is enough to guarantee the absence of immediate deadlock;

acyclicity at every sub-term ensures that this deadlock–freedom is preserved by reduction. While the session dependency graph approximates session interference and interdependency in a powerful and intuitive way, the acyclicity of the session dependency graphs is at heart a simple syntactic property, easily checkable in linear time by considering the free names of a process.

Related Work. Progress has been investigated for a variety of calculi for web services e.g., in [1, 3, 4] and, for session types, it has spawned several lines of research. The present paper takes as its starting points [9] and [7]. In particular, the former gives a progress result for a class of well-typed processes identified by a particular typing system based on finding an ordering of channel usage, in contrast to the session dependency graph ordering threads. Importantly, [9] allows service channels to be restricted, something that we do not presently. However, transparent processes are neither a subset nor a superset of the processes characterized in [9]. We clarify the key differences with some examples. First, the process

$$\mathtt{buy}(k).\,\overline{\mathtt{ship}}(k').\,k \triangleright \left\{ \begin{array}{rl} \mathtt{ok}: & k?(x_{\mathtt{addr}}).\,k'!\langle x_{\mathtt{addr}}\rangle, \\ \mathtt{abort}: & k'!\langle \mathtt{null}\rangle.\,k?(x_{\mathtt{reason}}) \end{array} \right\} \tag{1}$$

denotes a service `buy` which, upon invocation, creates the session channel k, then calls service `ship` and finally branches (with labels `ok` and `abort`) on k. The two branches differ by the order in which k and k' are used. This process is transparent, and thus has progress (in a context with a suitable invocation $\overline{\mathtt{buy}}(k)$ and service $\mathtt{ship}(k')$). However, because of the inconsistent orderings on the use of channels in the two branches (communication on k then k' in one, k' then k in the other), the typing of [9] rejects this process. Second, the process

$$\mathtt{buy}(k).\,k?(x_{\mathtt{card}}).\,\mathtt{serv}(k').\,k'!\langle 5\rangle \quad | \quad \overline{\mathtt{buy}}(k).\,\overline{\mathtt{serv}}(k').\,k'?(y).\,k!\langle \mathtt{card}\rangle \tag{2}$$

consists of the parallel composition of a service `buy` and a client. Service `buy`, expects to receive credit card details on its private channel k for a payment, and then spawns some service `serv` which sends some value (5 in this case). However, the client expects to pay after it has used the service i.e. it will invoke immediately serv after invoking `buy`. Payment is done eventually after the service is used. Notice that, if run on its own, the process will get stuck. However, if provided with the right context, it progresses simply because it will reduce in the presence of another suitable service `s`. Again, in [9], this term is rejected because of the inconsistent order in which `buy` and `s` are used. In this work, this process is transparent, and thus has progress.

Other approaches to progress for session types include [10, 8], which considers models featuring synchronous/asynchronous session types for object-oriented languages. In particular, [10] also exploits the assumption that a service cannot rely on already open communications. However, in that work, a delegation $k\langle\!\langle k'\rangle\!\rangle.\,P$ is well-typed only if P does not contain further uses of k. That is, a process can truly only ever participate in one session at a given time. In contrast, while we insist that services at the outset do not reference open sessions, a service may evolve through delegation to participate in many sessions at the same time. A case in point is (1) above.

In [2], a typing system ensures progress for multiparty session types where sessions may involve more than two participants. Unlike our result, the aforementioned works introduce extra typing for guaranteeing progress of programs. However, because of asynchronous communications, limitations as the one described in the previous paragraph are present only for processes receiving a delegated channel.

The present paper expands in several directions the session dependency graphs introduced in [7]. First, the graphs in [7] address a different language dealing with exceptions and do not handle higher-order sessions (delegation). Second, whereas [7] treated only programs, the present work takes the much larger class of transparent processes.

Contribution of the paper. The main technical contribution of this paper is the investigation of session interdependency graphs and the identification of a class of well-typed processes with progress, the *transparent* processes. This is an interesting class because:

1. it is potentially practically relevant: as explained above, it is a natural model of WS;
2. it is simply characterized by acyclicity of its session dependency graph, a syntactic condition, checkable in linear time. No special typing system is necessary;
3. it includes new processes not identified as progressing by known methods.

This contribution is relevant to programming languages based on session types, e.g., [12], as it gives a simple syntactic means for ensuring progress for protocols implemented in such languages.

Outline of the presentation. We present the π-calculus with sessions in § 2 and introduce a slight variation of session types in § 3. § 4 introduces session dependency graphs while § 5 defines the class of transparent processes, and prove that this class is closed under reduction. § 6 proves that every transparent process progresses, and, as a corollary, that so does every program. We conclude in § 7. For space reasons, some (parts of) proofs have been omitted and moved to the appendix of the online version [5].

2 A π-calculus with Service Oriented Sessions

We introduce a variant of the π-calculus with sessions [17, 11] which outlaws restriction on public channels, and allows replicated behaviour only for services.

Syntax. Let $a,b,c,x,y,z,\ldots$ range over *service (or public) channels*; $k,k',t,s,\ldots$ over *session (or private) channels*; and $e,e',\ldots$ over public channels, and arithmetic and other first-order *expressions*.

$P ::=$	0	(inact)	$\mid P \mid Q$	(par)
	$\mid (\nu k)\, P$	(resSess)	$\mid \gamma$	(prefix)
$\gamma ::=$	$!a(k).\, P$	(repServ)	$\mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$	(cond)
	$\mid a(k).\, P$	(serv)	$\mid \overline{a}(k).\, P$	(request)
	$\mid k?(x).\, P$	(input)	$\mid k!\langle e\rangle.\, P$	(output)
	$\mid k(\!(k')\!).\, P$	(inputS)	$\mid k\langle\!\langle k'\rangle\!\rangle.\, P$	(delegation)
	$\mid k \triangleright \{l_i : P_i\}_{i\in I}$	(branch)	$\mid k \triangleleft l.\, P$	(select)

Above, the class of prefixes γ includes services (serv), replicated services (repServ), and service invocations (request); as well as in-session communication (input, output), receive and send of session channels (inputS, delegation), and branching (branch, select). The other operators are standard. The free session (service) channels of a process P, denoted by $\mathsf{fsc}(P)$ ($\mathsf{fv}(P)$), are defined as usual. For the sake of simplicity, we have removed recursion. We conjecture that our results can be easily extended.

Example 1 (Buyer-Seller protocol)**.** We recall a variant of the Buyer-Seller protocol from [6] where a buyer invokes a service buy at a seller for a quote about some product. In case of acceptance by the buyer, the seller will place the order by invoking a service `ship` at a shipper and forward credit card details. Finally, the shipper will send directly a confirmation to the buyer. Such a protocol can be described by

(RINIT)	$!a(k).\, P \mid \overline{a}(k).\, Q \to !a(k).\, P \mid (\nu k)\,(P \mid Q)$	(INIT)	$a(k).\, P \mid \overline{a}(k).\, Q \to (\nu k)\,(P \mid Q)$
(COM)	$k?(x).\, P \mid k!\langle e\rangle.\, Q \to P[v/x] \mid Q \quad (e \Downarrow v)$	(DEL)	$k(\!(k')\!).\, P \mid k\langle\!\langle k'\rangle\!\rangle.\, Q \to P \mid Q$
(SEL)	$k \triangleright \{l_i : P_i\}_{i\in I} \mid k \triangleleft l_j.\, Q \to P_j \mid Q \quad (j \in I)$	(PAR)	$P \to P' \;\Rightarrow\; P \mid Q \to P' \mid Q$
(RES)	$P \to P' \;\Rightarrow\; (\nu k)\,P \to (\nu k)\,P'$	(STR)	$P \equiv Q,\ Q \to Q',\ Q' \equiv P' \;\Rightarrow\; P \to P'$
(IFT)	$\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \to P \quad (e \Downarrow \mathtt{tt})$	(IFF)	$\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \to Q \quad (e \Downarrow \mathtt{ff})$

Table 1: Reduction Semantics

the process $P_{\text{Buyer}} \mid P_{\text{Seller}} \mid P_{\text{Shipper}}$ such that:

$$
\begin{array}{rcl}
P_{\text{Buyer}} & \stackrel{\text{def}}{=} & \overline{\mathtt{buy}}(k).\ k?(x_{\mathtt{quote}}).\ \mathsf{if}\ x_{\mathtt{quote}} \leq 100\ \mathsf{then}\ k \triangleleft \mathtt{ok}.\, k?(x_{\mathtt{conf}}).\, 0\ \mathsf{else}\ k \triangleleft \mathtt{stop}.\, 0 \\
P_{\text{Seller}} & \stackrel{\text{def}}{=} & !\mathtt{buy}(k).\, k!\langle \mathtt{quote}\rangle.\ k \triangleright \{\ \mathtt{ok} : \overline{\mathtt{ship}}(k').\, k'(\!(k)\!).\, 0,\quad \mathtt{stop} : 0\ \} \\
P_{\text{Shipper}} & \stackrel{\text{def}}{=} & !\mathtt{ship}(k').\, k'(\!(k)\!).\ k!\langle \mathtt{conf}\rangle.\, 0
\end{array}
$$

Structural Congruence and Reduction Semantics. The structural congruence $\equiv$ is standard and is defined as the minimal relation satisfying the following rules:

$(i)\ P \mid Q \equiv Q \mid P$	$(ii)\ P \mid (Q \mid R) \equiv (P \mid Q) \mid R$		$(iii)\ P \mid 0 \equiv P$	
$(iv)\ P \equiv Q \quad (\text{if } P =_\alpha Q)$	$(v)\ P \mid (\nu k)\,Q \equiv (\nu k)\,(P \mid Q)$	$(k \notin \mathsf{fsc}(P))$	$(vi)\ (\nu k)0 \equiv 0$	

The standard reduction semantics $\to$ [11] is reported in Table 1 where $e \Downarrow v$, taking expressions to some values, is unspecified. Note that we have adopted the original (DEL) rule [11] which requires the receiving side to "guess" what is being delegated (as in internal π-calculus [15]). As a consequence, process $k(\!(k'')\!).\, P \mid k\langle\!\langle k'\rangle\!\rangle.\, Q$ fails to reduce if k' is free in P, as we cannot rename k'' to k' [19]. We shall see that such process violates transparency (see Remark 25).

We conclude this section with two auxiliary notions. First, the notion of program, i.e., a process containing no free session channels and no occurrences of restricted session channels.

Definition 2 (Program)**.** *A process P is a* program *whenever $\mathsf{fsc}(P) = \emptyset$ and there exists $P' \equiv P$ s.t. P' has no syntactic sub-term $(\nu k)\,Q$.*

Second, sub-processes, i.e., sub-terms of a process:

Definition 3 (Sub-Process)**.** *A process Q is a* sub-process *of P iff Q is a sub-term of some $P' \equiv P$.*

3 Session Typing

Syntax. Session types abstract the way a single session channel is used within a single session. The structure of a session is represented by a type, which is then used as a basis for validating protocols through an associated type discipline. Their syntax is given by the following grammar:

$$
\begin{array}{lll}
\alpha & ::= ?(\theta).\, \alpha \mid !(\theta).\, \alpha \mid \&\{l_i : \alpha_i\} \mid \oplus\{l_i : \alpha_i\} \mid \mathtt{end} & \\
\theta & ::= S \mid \alpha & S ::= \mathsf{basic} \mid \langle \alpha \rangle
\end{array}
$$

$$\text{(T-Serv)}\ \frac{\Gamma,\, a:\langle\alpha\rangle \vdash P \triangleright k:\alpha}{\Gamma,\, a:\langle\alpha\rangle \vdash a(k).\, P \triangleright \emptyset} \qquad \text{(T-Req)}\ \frac{\Gamma,\, a:\langle\alpha\rangle \vdash P \triangleright \Delta\cdot k:\overline{\alpha}}{\Gamma,\, a:\langle\alpha\rangle \vdash \overline{a}(k).\, P \triangleright \Delta}$$

$$\text{(T-In)}\ \frac{\Gamma,\, x:S \vdash P \triangleright \Delta\cdot k:\alpha}{\Gamma \vdash k?(x).\, P \triangleright \Delta\cdot k:?(S).\,\alpha} \qquad \text{(T-Out)}\ \frac{\Gamma \vdash P \triangleright \Delta\cdot k:\alpha \qquad \Gamma \vdash e:S}{\Gamma \vdash k!\langle e\rangle.\, P \triangleright \Delta\cdot k:!(S).\,\alpha}$$

$$\text{(T-InS)}\ \frac{\Gamma \vdash P \triangleright \Delta\cdot k:\alpha\cdot k':\beta}{\Gamma \vdash k(\!(k')\!).\, P \triangleright \Delta\cdot k:?(\beta).\,\alpha} \qquad \text{(T-Del)}\ \frac{\Gamma \vdash P \triangleright \Delta\cdot k:\alpha}{\Gamma \vdash k\langle\!\langle k'\rangle\!\rangle.\, P \triangleright \Delta\cdot k:!(\beta).\,\alpha\cdot k':\beta}$$

$$\text{(T-Bra)}\ \frac{\Gamma \vdash P_i \triangleright \Delta\cdot k:\alpha_i}{\Gamma \vdash k\triangleright\{l_i:P_i\}_{i\in I} \triangleright \Delta\cdot k:\ \&\{l_i:\alpha_i\}} \qquad \text{(T-Sel)}\ \frac{\Gamma \vdash P \triangleright \Delta\cdot k:\alpha_j}{\Gamma \vdash k\triangleleft l_j.\, P \triangleright \Delta\cdot k:\oplus\{l_i:\alpha_i\}}$$

$$\text{(T-Par)}\ \frac{\Gamma \vdash P_i \triangleright \Delta_i \qquad \Delta_1 \asymp \Delta_2}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1 \odot \Delta_2} \qquad \text{(T-Inact)}\ \frac{\alpha_i = \mathtt{end}}{\Gamma \vdash 0 \triangleright k_1:\alpha_1 \cdot \ldots \cdot k_n:\alpha_n}$$

$$\text{(T-RServ)}\ \frac{\Gamma,\, a:\langle\alpha\rangle \vdash P \triangleright k:\alpha}{\Gamma,\, a:\langle\alpha\rangle \vdash\ !a(k).\, P \triangleright \emptyset} \qquad \text{(T-Res)}\ \frac{\Gamma \vdash P \triangleright \Delta\cdot k:\bot}{\Gamma \vdash (\nu k)\, P \triangleright \Delta}$$

$$\text{(T-Cond)}\ \frac{\Gamma \vdash e:\mathsf{bool} \quad \Gamma \vdash P_i \triangleright \Delta}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \triangleright \Delta} \qquad \text{(T-Bot)}\ \frac{\Gamma \vdash P \triangleright \Delta\cdot k:\mathtt{end}}{\Gamma \vdash P \triangleright \Delta\cdot k:\bot}$$

Table 2: Typing Rules for Session Types

Here, $?(\theta).\,\alpha$ and $!(\theta).\,\alpha$ denote in-session input and output followed by the communications in α. The type θ abstracts what is communicated: a basic value (basic denotes basic types, e.g., int or bool), a service channel of type $\langle\alpha\rangle$, or a session channel of type α. Finally, $\&\{l_i:\alpha_i\}$ and $\oplus\{l_i:\alpha_i\}$ denote branching and selection types, and end is the inactive session. The *dual* of α, written $\overline{\alpha}$, is defined as

$$\begin{array}{lllllllll} \overline{?(\theta).\,\alpha} & = & !(\theta).\,\overline{\alpha} & \qquad & \overline{!(\theta).\,\alpha} & = & ?(\theta).\,\overline{\alpha} & & \\ \overline{\&\{l_i:\alpha_i\}} & = & \oplus\{l_i:\overline{\alpha}_i\} & \qquad & \overline{\oplus\{l_i:\alpha_i\}} & = & \&\{l_i:\overline{\alpha}_i\} & \qquad \overline{\mathtt{end}} & = & \mathtt{end} \end{array}$$

Environments, Judgements and Typing Rules. We define two typing environments, namely the *service* and the *session* typing.

$$\begin{array}{llllll} \textit{(Service Typing)} & \Gamma & ::= & \Gamma,\, a:S \mid \emptyset & & \\ \textit{(Session Typing)} & \Delta & ::= & \Delta\cdot k:\alpha \mid \Delta\cdot k:\bot \mid \emptyset & & \end{array}$$

The service typing fixes usage of service channels, whereas session typing fixes usage of session channels. In Δ, a session channel k may be assigned to $\bot$ rather than a session type α. This is to note that the two sides of session k have already been found in a sub-term (therefore k cannot be used further). When convenient, we treat both environments as functions mapping channels to their types.

Judgements have the form $\Gamma \vdash P \triangleright \Delta$ and are defined in Table 2. The rules are all standard [11] except from (T-Serv). As mentioned in the introduction, the fresh channel k is the *only* session channel present in the session environment for the subprocess P. (T-Bot) is necessary in order to guarantee subject congruence [19]. We remind the reader that, in (T-Par), $\Delta_1 \asymp \Delta_2$ (duality check) holds whenever $\Delta_1(k) = \overline{\Delta_2(k)}$ for every $k \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)$. Moreover, $\Delta_1 \odot \Delta_2$ (assign $\bot$ when both sides of a session have been found) is defined as $\Delta_1 + \Delta_2 + \bigcup_{k\in\mathsf{dom}(\Delta_1)\cap\mathsf{dom}(\Delta_2)} k:\bot$.

Example 4. The Buyer-Seller protocol from Example 1 is clearly well typed according to the rules in Table 2. In fact, the judgement $\Gamma \vdash P_{\text{Buyer}} \mid P_{\text{Seller}} \mid P_{\text{Shipper}} \triangleright \emptyset$ holds whenever Γ contains:

$$\begin{array}{rl} \texttt{buy}: & !(\texttt{int}).\,\&\{\,\texttt{ok}:!(\texttt{string}).\,\texttt{end},\quad \texttt{stop}:\texttt{end}\,\} \\ \texttt{ship}: & ?(!(\texttt{string})).\,\texttt{end} \end{array}$$

Clearly, a process well-typed according to Table 2 is also well-typed according to the standard typing [11]. As a consequence, it is straightforward to obtain the standard subject congruence/reduction results.

Theorem 5. *Let $\Gamma \vdash P \triangleright \Delta$. Then, (1) $P \equiv Q$ implies $\Gamma \vdash Q \triangleright \Delta$; and (2) $P \to P'$ implies $\Gamma \vdash P \triangleright \Delta$.*

4 Session Dependency Graphs

In this section, we recall, generalize, and develop *session dependency graphs*, introduced in [7] for a language dealing with exceptions and without delegation. In subsequent sections, we shall use them to establish progress for a class of well-typed processes.

Informally, given a restriction-free process $P \equiv \gamma_1 \mid \cdots \mid \gamma_n$ where each γ_i is called a "thread", P's session dependency graph has a node for each thread γ_i, and edges between threads γ_i, γ_j if the two share a free session channel. Observe that a thread γ_i can be blocked on a session channel k, waiting for γ_j to synchronize on k, *only if* the two have an edge between them. It follows that a well-typed process is deadlocked only if its session dependency graph contains a cycle (we prove this result formally in Theorem 23). Notice how this approach to deadlock detection differs from the focus on the ordering of the channels usage found frequently in the literature, e.g., in [13, 9].

Definition 6 (Session Dependency Graph)**.** *Let P be well-typed process. The* session dependency graph $\mathscr{G}(P) = (\mathscr{N}(P), \mathscr{E}(P), \mathscr{L}_P)$ *is the labelled unoriented graph*[1] *with nodes $\mathscr{N}(P)$, edges $\mathscr{E}(P)$ and labels $\mathscr{L}_P : \mathscr{N}(P) \to \mathscr{P}(N)$ defined inductively on the term structure of P as follows:*

$$\begin{array}{rcl} \mathscr{G}(0) & = & (\emptyset,\quad \emptyset,\quad \emptyset) \\ \mathscr{G}((\nu k)\,P) & = & (\,\mathscr{N}(P),\quad \mathscr{E}(P),\quad \mathscr{L}_P \setminus \{k\}\,) \\ \mathscr{G}(\gamma) & = & (\,\bullet,\quad \emptyset,\ [\bullet \mapsto \mathsf{fsc}(\gamma)]\,) \\ \mathscr{G}(P \mid Q) & = & (\,\mathscr{N}(P)+\mathscr{N}(Q),\quad \mathscr{E}(P)+\mathscr{E}(Q)+\Sigma_{\substack{p\in\mathscr{N}(P)\\ q\in\mathscr{N}(Q)\\ k\in\mathscr{L}_P(p)\cap\mathscr{L}_Q(q)}}(p,q),\quad \mathscr{L}_P+\mathscr{L}_Q\,) \end{array}$$

Here, we take $(\mathscr{L}_P \setminus \{k\})(p) = \mathscr{L}_P(p) \setminus \{k\}$.

The definition of graph is insensitive to structural congruence:

Lemma 7. *Let P, Q be well-typed processes s.t. $P \equiv Q$. Then $\mathscr{G}(P) \simeq \mathscr{G}(Q)$.*

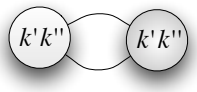
Proof. By cases on the definition of $\equiv$. □

Example 8. Consider the following process.

$$k'?(x).\,k''!\langle x\rangle \mid k''?(x).\,k'!\langle x\rangle \tag{3}$$

The session dependency graph of this process has two nodes, both labelled by $\{k', k''\}$, and thus has two edges between those two nodes:

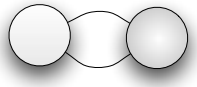
[1] Technically, a graph is here a 5-tuple $(N, E, L, \mathrm{dom}, \mathrm{cod})$, where the latter two are maps $\mathrm{dom}, \mathrm{cod} : E \to N$ taking edges to their domain and co-domain respectively. We elide these maps; their values shall always be clear from the context.



Thus, the graph has a cycle. Consider the same process, only restricting k', k''.

$$(\nu k'k'')\ (k'?(x).\ k''!\langle x\rangle \mid k''?(x).\ k'!\langle x\rangle) \tag{4}$$

Its graph has the same node and edges, but both nodes are now labelled by $\emptyset$.



Obviously, also this graph contains a cycle. Now, prefix the process with a service.

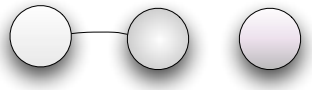
$$a(k).\ (\nu k'k'')\ (k'?(x).\ k''!\langle x\rangle \mid k''?(x).\ k'!\langle x\rangle) \tag{5}$$

This process has a graph with a single, unlabelled node and no edges which, obviously, contains no cycle:

Example 9. The graph of $P_{\text{Buyer}} \mid P_{\text{Seller}} \mid P_{\text{Shipper}}$ in Example 1 has clearly three unlabelled nodes. However, after one step of reduction, we have:

$$(\nu k)\left(\ k?(x_{\texttt{quote}}).\ Q_1 \ \mid\ k!\langle \texttt{quote}\rangle.\ Q_2\ \right) \mid P_{\text{Shipper}} \tag{6}$$

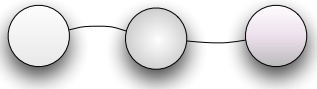
where Q_1 and Q_2 are the remainders of buyer and seller's processes. The session interdependency graph of the process above becomes (restriction removes labels):



If we further reduce (6) until also shipper is invoked, we obtain the process:

$$(\nu k,k')\left(\ k?(x_{\texttt{conf}}).\,0 \ \mid\ k'\langle\!\langle k\rangle\!\rangle.\,0 \ \mid\ k'(\!(k)\!).\,k!\langle \texttt{conf}\rangle.\,0\ \right)$$

whose graph is:



Notation. To ease the presentation, we shall frequently say, e.g., "p is a node of $\mathscr{G}(P)$" and "p is labelled by k in $\mathscr{G}(P)$" rather than the less readable "$p \in \mathscr{N}(P)$" and "$k \in \mathscr{L}_P(p)$". Moreover, when no confusion may arise, we drop the "in/of $\mathscr{G}(P)$", saying simply, e.g., "p is labelled k".

We conclude this section by noting two important facts about session dependency graphs. First, the combinatorics of typing.

Lemma 10. *Let $\Gamma \vdash P \triangleright k_1 : \bot \cdot \ldots \cdot k_n : \bot \cdot k'_1 : \alpha_1 \cdot \ldots \cdot k'_m : \alpha_m$. Then (1) for each $i \leq m$ at most one node p of $\mathcal{G}(P)$ is labelled k'_i, and (2) when R contains no top-level restrictions $\mathcal{G}(P)$ has at most n edges.*

Proof. By induction on R. □

Second, the structure of programs. Recall from Section 2 that programs have no free session channels and no non-trivial restricted session channels.

Proposition 11. *Let P be a well-typed program and Q any of its sub-process. Then $\mathcal{G}(Q)$ has no edges.*

Proof. Note $\Gamma \vdash P \triangleright \emptyset$ for some Γ. Let R be a sub-process of P. Then, also $\Gamma_R \vdash R \triangleright \Delta_R$ for some Γ_R, Δ_R. By Lemma 10, it is enough to prove that every $k \in \mathrm{dom}(\Delta_R)$ has $\Delta_R(k) \neq \bot$. We show that $\Delta_R(k) = \bot$ for some k would contradict $\Gamma \vdash P \triangleright \emptyset$. But for this, it is sufficient to prove that every rule preserves $k : \bot$ in Δ from premises to conclusion. This is trivial for all rules but (T-RES) and (T-SERV). Now, (T-RES) does not apply to programs and their sub-processes, and (T-SERV) accepts no $k : \bot$ in its premises. □

5 Transparent Processes

In this section, we define the notion of *transparent process* and investigate its properties. In particular, we prove that every program is transparent and that transparent processes are closed under reduction. Along the way, we get to thoroughly exercise session dependency graphs, exhibiting their particular strengths. These results pave the way for proving that every transparent process has progress in the next section.

We define the transparent processes as those whose graphs are, essentially, "everywhere acyclic".

Definition 12 (Transparent Process)**.** *Let P be a well-typed process. We say that P is* transparent *iff every sub-process Q of P has $\mathcal{G}(Q)$ acyclic.*

Example 13. Neither the process of (3), of (4) nor of (5) are transparent. The former two because they themselves have cyclic session dependency graphs, the latter because it contains a sub-process which has (namely (4)). However, the buyer-seller system from Example 1 is transparent.

Transparent processes are closed under structural congruence:

Lemma 14. *Let P, Q be well-typed processes s.t. $P \equiv Q$. Then P transparent iff Q transparent.*

Proof. By cases on the definition of $\equiv$. □

The rest of this section is dedicated to proving our first main result, i.e., transparency is preserved by reduction. In the next section, we will use these results to prove that transparent processes progress.

In order to see that a reduction $P \to P'$ preserves transparency, we need to relate the session dependency graph of P to the one of P'. Informally, we make the following observations:

1. The set of free session channels of a process is non-increasing under reduction. Thus, even though the nodes and edges of $\mathcal{G}(P)$ and $\mathcal{G}(P')$ might be very different, the *labels* of $\mathcal{G}(P')$ contain only names also in labels of $\mathcal{G}(P)$.
2. We can speak of a label k of $\mathcal{G}(P)$ having a path to a label k' in both graphs if there is a path from a node labelled k to a node labelled k'. Thus, we arrive at the invariant: if k has a path to k' in $\mathcal{G}(P')$, it also had so in $\mathcal{G}(P)$. This property is enough to ensure transparency preservation.

We proceed to define precisely this notion of a path from k to k'.

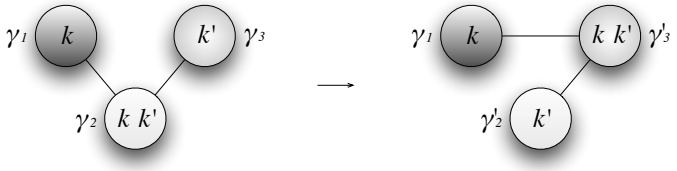
Definition 15. *Let P be a well-typed process, and let $k,k' \in \mathsf{fsc}(P)$. We say that "$k \rightsquigarrow k'$ in $\mathscr{G}(P)$" iff there exist nodes p,p' in $\mathscr{G}(P)$, labelled by k,k' respectively, s.t. there is a path from p to p' in $\mathscr{G}(P)$.*

Observe that the "$- \rightsquigarrow -$" relation is reflexive (because k,k' need not be distinct and there is always a path from a node to itself) and symmetric (because the graph is). Moreover, because any two nodes p,p' sharing a label k will have an edge between them, we could equivalently have defined "$k \rightsquigarrow k'$ iff *for any two* nodes p,p' labelled by k,k', there is a path from p to p'." It follows that $- \rightsquigarrow -$ is transitive.

Remark 16. One of the key insights carrying the proof that reduction preserves transparency is that $- \rightsquigarrow -$ is non-increasing under reduction of transparent processes. It is instructive to see this for a particular case involving delegation. Consider the following reduction:

$$\underbrace{k!\langle 5\rangle}_{\gamma_1} \mid \underbrace{k'\langle\!\langle k\rangle\!\rangle.\, k'?(x)}_{\gamma_2} \mid \underbrace{k'(\!(k)\!).\, k?(x).\, k'!\langle 7\rangle}_{\gamma_3} \quad\to\quad \underbrace{k!\langle 5\rangle}_{\gamma_1} \mid \underbrace{k'?(x)}_{\gamma_2'} \mid \underbrace{k?(x).\, k'!\langle 7\rangle}_{\gamma_3'}$$

Before reduction, γ_1 and γ_2 share session k, and γ_2 and γ_3 share the session k'. After reduction, session k has moved, and is now between γ_1 and γ_3. Graphically, the session dependency graphs change as follows:



However, as it is immediately obvious from the graphical representation, connectivity of k and k' did not change: they met at exactly one node before reduction, and they meet at exactly one node after reduction. This is the essential reason why delegation cannot introduce a cycle in the session dependency graph.

Theorem 17. *[Preservation of transparency] Let R be a well-typed process s.t. $R \to R'$. Then*

1. *If $\mathscr{G}(R)$ is transparent then so is $\mathscr{G}(R)'$, and*
2. *if $k \not\rightsquigarrow k'$ in R, then also $k \not\rightsquigarrow k'$ in R'.*

Before the proof, observe that part (2) is vacuously true for $k,k' \notin \mathsf{fsc}(R)$.

Proof (sketch). Technically, the proof proceeds by induction on the derivation of $R \to R'$. Hereby, we discuss the most interesting cases.

- (Init) $\quad a(k).\, P \mid \overline{a}(k).\, Q \to (\nu k)\ (P \mid Q)$.
  1. Because $\mathscr{G}(a(k).\, P \mid \overline{a}(k).\, Q)$ is transparent, so are $\mathscr{G}(P)$ and $\mathscr{G}(Q)$. Thus, in order to prove $\mathscr{G}((\nu k)\ (P \mid Q))$ transparent, it is sufficient to prove it acyclic. By (T-Serv), there exists α s.t. $\Gamma \vdash P \triangleright k : \alpha$. Thus, by Lemma 10, the nodes of $\mathscr{G}(P)$ have empty labelling, except for at most one node; and that this unique node is labelled k (if it exists at all). Now note that there exists also Δ' s.t. $\Gamma \vdash Q \triangleright \Delta' \cdot k : \overline{\alpha}$. By linearity of the session environment, and Lemma 10, we find that at most one node of $\mathscr{G}(Q)$ is labelled k. Thus $\mathscr{G}((\nu k)\ (P \mid Q))$ is formed by adding at most one edge between acyclic graphs $\mathscr{G}(P)$ and $\mathscr{G}(Q)$ and is hence itself acyclic.
  2. It is sufficient to prove $k' \rightsquigarrow_R k''$ for any $k',k'' \in \mathsf{fsc}(R)$. Observe that $\mathscr{G}(a(k).\, P \mid \overline{a}(k).\, Q)$ has exactly two nodes, one in $\mathscr{G}(a(k).\, P)$ and one in $\mathscr{G}(\overline{a}(k).\, Q)$. As $\mathsf{fsc}(a(k).\, P)$ is empty by typing, it follows that the free session channels of the R are $\mathsf{fsc}(\overline{a}(k).\, Q)$. But then the node $\mathscr{G}(\overline{a}(k).\, Q)$ is labelled by every free session channel name of R, whence trivially, $k' \rightsquigarrow_R k''$.

- (COM) $k?(x).\,P \mid k!\langle e\rangle.\,Q \to P[v/x] \mid Q \qquad (e \Downarrow v)$.
  1. As $\mathscr{G}(R) = \mathscr{G}(k?(x).\,P \mid k!\langle e\rangle.\,Q)$ is transparent then also $\mathscr{G}(P)[v/x]$ and $\mathscr{G}(Q)$ are transparent (the former because neither v nor x are session channel names). It is now sufficient to prove that the graph $\mathscr{G}(R') = \mathscr{G}(P[v/x] \mid Q)$ is acyclic. Now, assume for a contradiction that $\mathscr{G}(R')$ contains a cycle. Because $\mathscr{G}(P)$ and $\mathscr{G}(Q)$ are transparent and thus acyclic, it must be the case that there exist two distinct edges connecting $\mathscr{G}(P)$ and $\mathscr{G}(Q)$. Suppose wlog that these edges arise from k',k'' with $k',k'' \in \mathsf{fsc}(P[v/x]) \cap \mathsf{fsc}(Q)$, and $k' \rightsquigarrow_P k'' \rightsquigarrow_Q k'$. As x is not a session channel name, so also $k',k'' \in \mathsf{fsc}(P)$. But then also $k',k'' \in \mathsf{fsc}(k?(x).\,P)$ and $k',k'' \in \mathsf{fsc}(k!\langle e\rangle.\,Q)$, so a cycle is already in $\mathscr{G}(R)$. Contradiction.
  2. We prove again that $k' \rightsquigarrow k''$ in $\mathscr{G}(R)$ for any $k',k'' \in \mathsf{fsc}(R)$. Observe again that $\mathscr{G}(R) = \mathscr{G}(k?(x).\,P \mid k!\langle e\rangle.\,Q)$ has exactly two nodes, this time with an edge between them induced by k. But then $\mathscr{G}(R)$ is in fact connected, so trivially $k' \rightsquigarrow k''$ in $\mathscr{G}(R)$.
- (DEL) $k(\!(k')\!).\,P \mid k\langle\!\langle k'\rangle\!\rangle.\,Q \to P \mid Q$
  1. By assumption $R = k(\!(k')\!).\,P \mid k\langle\!\langle k'\rangle\!\rangle.\,Q$ transparent, so also P,Q is transparent. It is now sufficient to prove that $P \mid Q$ is itself acyclic. Suppose it is not. Both P,Q transparent and hence acyclic, so $P \mid Q$ cyclic must mean that there exists $k'',k''' \in \mathsf{fsc}(P) \cap \mathsf{fsc}(Q)$. We must have, $k' = k''$ or $k' = k'''$, or R is itself cyclic; assume $k' = k'''$. Thus $k',k'' \in \mathsf{fsc}(P) \cap \mathsf{fsc}(Q)$. But because R is well-typed, we must have $k' \notin \mathsf{fsc}(Q)$; contradiction.
  2. Identical to the (COM) case.
- (PAR) $P \to P' \quad \Rightarrow \quad P \mid Q \to P' \mid Q$.
  1. Because $P \mid Q$ transparent, also P,Q transparent. By induction hypothesis, also P' transparent, so it is sufficient to prove $P' \mid Q$ acyclic. Assume for a contradiction that $P' \mid Q$ contains a cycle. As P',Q both acyclic, there must exist two edges between $\mathscr{G}(P')$ and $\mathscr{G}(Q)$. Assume wlog that these edges are induced by distinct $k,k' \in \mathsf{fsc}(P') \cap \mathsf{fsc}(Q)$ with $k \rightsquigarrow_{P'} k' \rightsquigarrow_Q k$. Because $P \to P'$ implies $\mathsf{fsc}(P') \subseteq \mathsf{fsc}(P)$, we find $k,k' \in \mathsf{fsc}(P)$, so by induction hypothesis, $k \rightsquigarrow_P k'$, and by composition $k \rightsquigarrow_P k' \rightsquigarrow_Q k$, and a cycle is already in $P \mid Q$. Contradiction.
  2. We prove the contrapositive: Supposing $k \rightsquigarrow_{P'|Q} k'$ for $k,k \in \mathsf{fsc}(P \mid Q)$, we will see that $k \rightsquigarrow_{P|Q} k'$. There must exist a sequence $k = k_0,\ldots,k_n = k'$ with each i having either $k_i \rightsquigarrow_{P'} k_{i+1}$ or $k_i \rightsquigarrow_Q k_{i+1}$. By the induction hypothesis, each $k_i \rightsquigarrow_{P'} k_{i+1}$ implies $k_i \rightsquigarrow_P k_{i+1}$, so, by stringing the path back together and noting that $\rightsquigarrow$ is transitive, we find $k \rightsquigarrow_{P|Q} k'$. □

6 Progress

In this section, we give our main technical results: that every transparent process has progress; and that all programs are transparent. Intuitively, a process has *progress* if it cannot reduce to a process that is "stuck". We shall follow [9] in taking a non-trivial process to be stuck if it is irreducible in any context. Thus, we do not consider a process stuck if it needs a service or a counter-party to an active session. In the sequel, a process contains no live session channels whenever all its session channels are bound by (repServ) or (serv). Moreover, let $E[\cdot]$ denote a reduction context, i.e., $E[\cdot] ::= \cdot \mid E[\cdot] \mid P \mid (\nu k)\,E[\cdot]$.

Definition 18 (Progress)**.** *A process P has progress if $P \to^* P'$ implies that for every reduction context $E[\cdot]$ with $P' \equiv E[P'']$, whenever P'' contains live channels then there exists a process Q s.t.*

(a)	$Q \not\to$	*(c)*	$P'' \mid Q \to R$
(b)	$P'' \mid Q$ *is well-typed*	*(d)*	R *has progress*

Observe that processes with no live channels have progress.

This very strong but somewhat intentional definition of progress captures the intuition that "a stuck process is one where no thread is permanently blocked". For a process P to have progress, any process P' reachable from P must be either without any live channel or such that any of its top-level sub-processes P'' can be composed in parallel with a process Q and: (a) Q is stuck; (b) $P'' \mid Q$ is well typed, i.e., Q only provides services, requests or counter-parties to active sessions in P''; and (c,d) $P'' \mid Q$ can reduce to a process that also has progress. The focus on sub-processes is to make sure that non-terminating processes do not automatically have progress, e.g.,:

$$k'?(x).\, k''!\langle x\rangle \;\mid\; k''?(x).\, k'!\langle x\rangle \;\mid\; !a(k).\, \overline{a}(k) \;\mid\; \overline{a}(k) \tag{7}$$

Remark 19. The definition above differs somewhat from the ones found in the literature, e.g., [9, 2]. The present one has the distinct advantage that is independent of the means chosen to *establish* progress. Other works use a special typing system to *establish* progress already in the *definition* of progress itself. For instance, the processes (1) and (2) on page 3, intuitively have progress: all of their sessions run to termination when given access to appropriate services. And, both processes have progress wrt our definition. However, neither has progress as defined in [9], where the definition of progress is inextricably linked to the typing system guaranteeing it, and both of those processes are untypable in that system (our definition requires typability for guaranteeing in-session linearity but it does not require transparency). As a further example, write P_1 for the $k'?(x).\, k''!\langle x\rangle \mid k''?(x).\, k'!\langle x\rangle$ and P_2 for the $k'?(x).\, k''!\langle x\rangle \mid k'!\langle e\rangle.\, k''?(x)$ in

$$k \triangleright \{ \quad l_1 : P_1 \quad l_2 : P_2 \quad \} \quad \mid \quad k \triangleleft l_2$$

Again, this process intuitively has progress — it can only run to termination —, it is included by the present definition of progress, but not by the one of [9]. Note that the process above is not transparent.

Having established that transparency is preserved by reduction, we proceed to show that a well-typed, transparent process has progress. In particular, we need to show that we can build the additional process Q found in the definition of progress. Our idea is to do that by exploiting the type of a given process. The next result shows that every session type is in fact inhabited, i.e., from a session type α we can reconstruct a process which behaves exactly as specified by α. In the sequel, we shall assume that all basic types are inhabited. For convenience, we further assume that they are all inhabited by the same value "1"; however this assumption is easily rendered unnecessary.

Lemma 20 (Every session type is inhabited)**.** *For any session type α and session channel k, there exists a transparent process $[\![\alpha]\!]^k$ with $\vdash [\![\alpha]\!]^k \triangleright k : \alpha$, defined in Figure 1.*

In the above lemma, the first three cases reconstruct a process that inputs a value, a service channel and a session respectively. (OUTVAL), (OUTSERV) and (OUTSESS) are their dual counterpart. (INSUM) and (OUTSUM) are external and internal choice while (END) generates the inactive process from the end type.

Example 21. Here is an illustrative example.

$$[\![?(\ ?(\mathsf{basic}).\ !(\mathsf{basic})\ ).\ !(\mathsf{basic})]\!]^k = k(\!(k')\!).\, (k!\langle 1\rangle \mid k'?(x).\, k'!\langle 1\rangle)$$

Note how received values are never used and how the process only ever sends "1".

(INVAL)	$[\![?(\mathsf{basic}).\ \alpha]\!]^k = k?(x).\ [\![\alpha]\!]^k$
(INSERV)	$[\![?(\langle\beta\rangle).\ \alpha]\!]^k = k?(a).\ [\![\alpha]\!]^k$
(INSESS)	$[\![?(\beta).\ \alpha]\!]^k = k(\!(k')\!).\ ([\![\alpha]\!]^k \mid [\![\beta]\!]^{k'})$
(OUTVAL)	$[\![!(\mathsf{basic}).\ \alpha]\!]^k = k!\langle 1\rangle.\ [\![\alpha]\!]^k$
(OUTSERV)	$[\![!(\langle\beta\rangle).\ \alpha]\!]^k = k!\langle a\rangle.\ ([\![\alpha]\!]^k \mid !a(k').\ [\![\beta]\!]^{k'})$
(OUTSESS)	$[\![!(\beta).\ \alpha]\!]^k = (\nu k')\ (k\langle\!\langle k'\rangle\!\rangle \mid [\![\alpha]\!]^k \mid [\![\overline{\beta}]\!]^{k'})$
(INSUM)	$[\![\&\{l_i : \alpha_i\}]\!]^k = k \triangleright \{l_i : [\![\alpha]\!]^k{}_i\}_{i\in I}$
(OUTSUM)	$[\![\oplus\{l_i : \alpha_i\}]\!]^k = k \triangleleft l_1.\ [\![\alpha_1]\!]^k$
(END)	$[\![\mathtt{end}]\!]^k = 0.$

Figure 1: Translation $[\![-]\!]^-$

Example 22. In the buyer-seller protocol, the session type of the session `buy` reported in Example 4 is $!(\mathtt{int}).\ \&\{\ \mathtt{ok} : !(\mathtt{string}).\ \mathtt{end},\quad \mathtt{stop} : \mathtt{end}\ \}$. According to the construction from the previous Lemma, for some k, we can build the process $k!\langle 1\rangle.\ k \triangleright \{\ \mathtt{ok} : k!\langle\text{"}a\text{"}\rangle.\ 0,\quad \mathtt{stop} : \mathtt{end}\ \}$, where we have chosen to inhabit `string` by the string "a".

Knowing that every session type is inhabited, we can prove that well-typed, transparent processes either have no live channels or can reduce given the proper environment.

Theorem 23 (Reduction of Transparent Processes)**.** *Let P be a well-typed, transparent process. Either P has no live channels, or there exists some $Q \not\rightarrow$ such that $P \mid Q$ is well-typed, transparent and $P \mid Q \rightarrow$.*

Proof. If P has no live channels then we are done. Also, if $P \rightarrow$ then also $P \mid 0 \rightarrow$, $0 \not\rightarrow$ trivially and clearly $P \mid 0$ transparent. So assume $P \not\rightarrow$. Assume wlog $P \equiv (\nu\tilde{k})\ (\gamma_1 \mid \cdots \mid \gamma_n)$ and $\Gamma \vdash P \triangleright \Delta$.

Suppose first that for some i, γ_i is a service invocation $\gamma_i \equiv \overline{a}(k).\ Q$. Then, by Lemma 20 there exists some Γ' with $\Gamma, \Gamma' \vdash P \mid a(k).\ [\![\Gamma(a)]\!]^k \triangleright \Delta$; clearly $P \mid a(k).\ [\![\Gamma(a)]\!]^k$ is transparent and reduces.

Suppose instead that no γ_i is such a service invocation. Now consider the case where for some k, $k : \alpha \in \Delta$. Clearly $k \notin \tilde{k}$, so again by Lemma 20, $P \mid [\![\overline{\alpha}]\!]^k$ is well-typed and transparent, and this process clearly reduces. So, consider instead (and finally) the case where every k mentioned by Δ in fact has $k : \bot \in \Delta$. We shall arrive at a contradiction, demonstrating that this typing is not possible for P transparent with $P \not\rightarrow$. Observe first that, up to labels, $\mathscr{G}(P) \simeq \mathscr{G}(\gamma_1 \mid \cdots \mid \gamma_n)$. We shall treat each γ_i interchangeably as a process and as a node of this graph. Suppose wlog that no γ_i has no live channels (otherwise, observe that γ_i would contain no free session channels, whence we may conduct the following argument in the sub-graph of those γ_i that have live channels). Thus, each γ_i has an enabled action on some k_i. Because $P \not\rightarrow$ and P well-typed, the k_i are pairwise distinct. Because P well-typed, for each i, there exists a unique j s.t. $k_i \in \mathsf{fsc}(\gamma_j)$. But then $\mathscr{G}(P)$ has n nodes and at least n edges, and must thus contain a cycle, contradicting transparency of P. □

Theorem 24. *Well-typed and transparent processes have progress.*

Proof. Suppose P is well-typed and transparent. Moreover, let $P \rightarrow^* P'$ and $P' \equiv E[P'']$. Observe that by definition also P'' is well-typed and transparent. Now, if P'' has no live channels, we are done. Otherwise, by Theorem 23, there exists Q s.t. $P'' \mid Q$ is transparent and well-typed, and $P'' \mid Q \rightarrow R$ for some R. By Theorem 5.2, this R is also well-typed, so by Theorem 17, R is also transparent. □

Remark 25. In Section 2, we discussed that rule (DEL) can cause a deadlock with a processes of the form $k(\!(k'')\!).\ P \mid k\langle\!\langle k' \rangle\!\rangle.\ Q$. However, this process is not transparent because its graph is not acyclic.

Recall from Section 2 that a program is a process with no free session channels and no non-trivial occurrences of session channel restrictions.

Corollary 26. *Well-typed programs have progress.*

Proof. By Proposition 11, every well-typed program is a transparent process. By the preceding Theorem, every transparent process has progress. □

Example 27. Example 1, buyer-seller, is a program, and so transparent, whence it has progress.

We conclude by remarking on the complexity of the implied program analysis. If a program is well-typed, transparency and thus progress comes *for free* courtesy of Corollary 26. It can be checked whether a process is a well-typed program in $\mathcal{O}(n)$, where n measures the number of nodes in the abstract syntax of the process; even if we have to also ensure that the process contains no restrictions.

The broader class of transparent processes has progress by Theorem 23. The session-dependency graph of a process is computable in $\mathcal{O}(cp)$ where c is the number of distinct service-channels, bound or free, and p is the number of top-level prefixes. Deciding acyclicity is linear in the number of nodes and edges, thus in time $\mathcal{O}(p+c) \subseteq \mathcal{O}(pc)$. Transparency requires this computation at every sub-term of the graph; however, it is clearly sufficient to consider every maximal parallel product sub-term (i.e., for $P \mid Q$, no need to consider P, Q separately). Thus, letting p_i be the width of the ith maximal parallel sub-term (that is, n for $P_1 \mid \cdots \mid P_n$), we can compute transparency in $\mathcal{O}(\Sigma_i p_i c) \subseteq \mathcal{O}(nc)$. In summary:

Theorem 28. *A well-typed process P can be checked for transparency in time $\mathcal{O}(nc)$, where n is the number of nodes in the abstract syntax of the process, and c is the number of distinct live channels.*

7 Conclusions

We have provided a simple and *efficient* static analysis for guaranteeing progress for web services based on session types. The advantage of our approach is that standard session typing (with the restriction on the typing of services) is enough for guaranteeing progress of programs without any further analysis of processes. Our result is based on the development of a technique which relies on session interdependency graphs. In particular, we have shown that transparent processes, those processes with an acyclic session interdependency graph, have progress based on the fact that transparency is preserved by the reduction semantics and it guarantees that live channels eventually react.

The main limitation of this work is the lack of service channel restriction. The main challenge with introducing such a syntactic construct is in the definition of progress. In fact, processes such as $(\nu a)\ \big(\overline{a}(k').\ k?(x) \mid\ !a(k')\big)$ should satisfy the progress property. However, our current definition would address the sub-process $\overline{a}(k').\ k?(x)$ which, taking restriction into account, has no progress. We leave this issue as future work conjecturing that, under some small assumptions, transparent processes with service restriction also have progress. Additionally, we plan to address two further points. Firstly, the graph representation of session interdependency has proved to be a very useful tool for investigating the properties of a system. We believe that this approach can lead to further results that can go beyond applications to progress e.g. secure data flow. Secondly, the work in [2] provides a typing system for guaranteeing progress in multiparty sessions. A natural question is to investigate whether the techniques used in this work can be reused in the multiparty session setting with similar results. We are optimistic that this might be the case.

Acknowledgments. We gratefully acknowledge helpful discussions with N. Yoshida, M. Dezani, K. Honda, and the anonymous referees.

References

[1] Lucia Acciai & Michele Boreale (2008): *A Type System for Client Progress in a Service-Oriented Calculus*. In: *Concurrency, Graphs and Models. Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, number 5065 in LNCS, Springer, pp. 642–658.

[2] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini & Nobuko Yoshida (2008): *Global Progress in Dynamically Interleaved Multiparty Sessions*. In: *19th International Conference on Concurrency Theory (Concur'08)*, LNCS, Springer, pp. 418–433.

[3] Roberto Bruni & Leonardo Gaetano Mezzina: *A deadlock free type system for a calculus of services and sessions*. Draft available online.

[4] Luis Caires & Hugo Vieira (2009): *Conversation Types*. In: *18th European Symposium on Programming (ESOP'08)*, *LNCS* 5502, Springer, pp. 285–300.

[5] Marco Carbone & Søren Debois (2010): *A Graphical Approach to Progress for Structured Communication in Web Services*. Online version of this paper available at `http://www.itu.dk/~maca/papers/CD10.pdf`.

[6] Marco Carbone, Kohei Honda & Nobuko Yoshida (2007): *Structured Communication-Centred Programming for Web Services*. In: *16th European Symp. on Programming (ESOP'07)*, *LNCS* 4421, Springer, pp. 2–17.

[7] Marco Carbone, Kohei Honda & Nobuko Yoshida (2008): *Structured Interactional Exceptions for Session Types*. In: *19th Int'l Conference on Concurrency Theory (Concur'08)*, LNCS, Springer, pp. 402–417.

[8] Mario Coppo, Mariangiola Dezani-Ciancaglini & Nobuko Yoshida (2007): *Asynchronous Session Types and Progress for Object-Oriented Languages*. In: *FMOODS'07*, *LNCS* 4468, pp. 1–31.

[9] Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro & Nobuko Yoshida (2007): *On Progress for Structured Communications*. In: *Trustworthy Global Computing (TGC'07)*, *LNCS* 4912, Springer, pp. 257–275.

[10] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida & Sophia Drossopoulou (2006): *Session Types for Object-Oriented Languages*. In: *Proceedings of ECOOP'06*, LNCS, Springer, pp. 328–352.

[11] Kohei Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Disciplines for Structured Communication-based Programming*. In: *7th European Symposium on Programming (ESOP'98)*, *LNCS* 1381, Springer-Verlag, pp. 22–138.

[12] Raymond Hu, Nobuko Yoshida & Kohei Honda (2008): *Session-Based Distributed Programming in Java*. In: Jan Vitek, editor: *ECOOP*, 5142, Springer, pp. 516–541.

[13] Naoki Kobayashi (2006): *A New Type System for Deadlock-Free Processes*. In: *CONCUR'06*, *LNCS* 4137, pp. 233–247.

[14] Riccardo Pucella & Jesse Tov (2008): *Haskell Session Types with (Almost) No Class*. In: *Proc. of the 1st ACM SIGPLAN Symposium on Haskell (Haskell'08)*, ACM SIGPLAN, pp. 25–36.

[15] Davide Sangiorgi (1996): *pi-Calculus, Internal Mobility, and Agent-Passing Calculi*. *Theoretical Computer Science* 167(1&2), pp. 235–274.

[16] Scribble (2008). *Scribble Project*. `www.scribble.org`.

[17] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-based Language and its Typing System*. In: *PARLE'94*, *LNCS* 817, Springer-Verlag, pp. 398–413.

[18] W3C. *World-Wide Web Consortium*. http://www.w3c.org.

[19] Nobuko Yoshida & Vasco Thudichum Vasconcelos (2007): *Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication*. *ENTCS* 171(4), pp. 73–93.