Assembling Sessions*

Philippe Darondeau[1], Loïc Hélouët[1], and Madhavan Mukund[2]

[1] IRISA, INRIA, Rennes, France
[2] CMI, Chennai, India
philippe.darondeau@irisa.fr, loic.helouet@irisa.fr, madhavan@cmi.ac.in

Abstract. Sessions are a central paradigm in Web services to implement decentralized transactions with multiple participants. Sessions enable the cooperation of workflows while at the same time avoiding the mixing of workflows from distinct transactions. Languages such as BPEL, ORC, AXML that implement Web Services usually realize sessions by attaching unique identifiers to transactions. The expressive power of these languages makes the properties of the implemented services undecidable. In this paper, we propose a new formalism for modelling web services. Our model is session-based, but avoids using session identifiers. The model can be translated to a dialect of Petri nets that allows the verification of important properties of web services.

1 Introduction

Web services consist of interactions between multiple parties. In developing a formal model for web services, we have to consider two different points of view. The first focus is on the interactions themselves: they are typically structured using what we will call *sessions*. An example of a session could include sending an email or making an online payment. Informally, a session is a functionally coherent sequence of interactions between agents playing specific roles, such as server and client.

The second requirement is to capture the perspective of each agent. Typically, agents participate in more than one session at a time: while composing a mail, an agent may also participate in an online chat and, on the side, browse a catalogue to select an item to purchase from an online retailer. While some concurrent sessions may be independent of each other, there may also be non-trivial connections between sessions. For instance, to purchase an item online, one has to first participate in a session with the retailer to choose an item, then make the online payment in a session with the bank, which typically returns the agent to the shopping session with a confirmation of the transaction. Thus, we need a mechanism to describe how an agent moves between sessions, including the possibility of invoking multiple concurrent sessions.

We propose a formal model for sessions to capture both these aspects. A guiding principle is that the model should support some formal verification. We base our approach on finite automata and model interaction through shared

* This work was partially funded by INRIA's DST Associated Team, and by the ARCUS program (Région Ile-de-France).

actions. These shared actions can update local variables of agents, which permits information to be transferred across agents. The local variables record the state of an agent across sessions to permit coordination between sessions.

Our model can be translated into a class of Petri nets called *Reset Post-G* nets [6] for which coverability is decidable. In terms of our model, this means, for instance, that asking whether a specific type of session occurs is decidable. The paper is organized as follows. After briefly discussing related work, we introduce our model through an example in the next section. This is followed by a formal definition of our model of session systems. Section 4 translates the semantics of session systems into Reset Post-G nets, and highlights decidability results for our model. We end with a brief conclusion. Due to lack of space, and also to improve readability, some proofs are only sketched.

Related work Several other frameworks propose sessions and mechanisms to *orchestrate* sessions into larger applications. The range of approaches includes agent-centric formalisms, such as BPEL [3], workflow-based formalisms such as Orc [8, 9], and declarative, rule-based formalisms such as AXML [1, 2]. Each approach has its advantages and drawbacks.

A BPEL specification describes a set of independent communicating agents equipped with a rich set of control structures. Coordination across agents is achieved through message-passing. Interactions are grouped into sessions implicitly by defining *correlations* which specify data values that uniquely identify a session—for instance, a purchase order number for an online retail transaction. This makes it difficult to identify the structure of sessions from the specification, and workflows are often implicit, known only at runtime. Orc is a programming language for the orchestration of services. It allows any kind of algorithmic manipulation of data, with an orchestration overlay that helps start new services and synchronize their results. Orc has better mechanisms to define workflows than BPEL, but lacks the notion of correlation that is essential to establish sessions among the participants in a service. AXML defines web services as a set of rules for transforming semi-structured documents described, for instance, in XML. However, it does not make workflows explicit, and does not have a native notion of session either. So, transactions must be defined using complex guards. A common feature of these formalisms is that they aim to describe *implementations* of web services or orchestrations. BPEL, Orc and AXML can easily simulate Turing Machines, hence rendering undecidable simple properties such as the termination of a service. In [4], the authors develop a model for *shared experience services* where multiple users participate in sessions by simultaneously accessing a shared communication interface—examples include conference calls and internet chat. Though this model has a superficial resemblance to our work—they define session data types and use finite automata to describe session behaviours—the main aim is to provide an event-driven programming language to describe such systems, without any support for verification, so their model has little in common with ours.

At a different level, Petri nets have often been used to specify workflows [10] or to serve as targets for translating high-level description languages such as BPEL [7, 11]. However, Petri nets are not expressive enough to model sessions

with correlations as in BPEL, hence translations either are restricted to a subset of the workflow description language [11] or they are aimed at coloured extensions of Petri nets [7] for which many properties are undecidable. Our model can be translated to a less powerful class of Petri Nets which allows us to decide properties such as coverability and termination.

2 Motivational example

To motivate the constructs that we incorporate into our model, we look at an example. We model an online retail system with three types of participants: clients (the buyers), servers (the sellers), and banks. The interactions between these entities can be broken up into two distinct phases: selecting and confirming the items to be purchased online, and paying for these items. The first phase, online sale, only concerns clients and servers while the second phase, online payment, involves all three types of entities.

In an online sale, a client logs in to a server and selects a set of items to buy. Selecting an item involves browsing the items on offer, choosing some of them, perhaps revoking some earlier choices and finally deciding to pay for the selected items. At this point, the client has to choose between several modes of payment. Once this choice is made, the online sale interaction is suspended and the second phase is triggered.

The second phase, online payment, involves the client and the server as well as a bank that is chosen by the server according to the mode of payment selected by the client. The server transfers the transaction amount to the bank. The bank then asks the client for credentials to authenticate itself and authorize this transaction. Based on the information provided by the client, the bank either accepts or rejects the transaction. This decision is based on several parameters, including the correctness of the authentication data provided and the client's credit limit. For simplicity, we can omit the details of how the bank arrives at this decision and model this as a nondeterministic choice between success and failure of the payment. When the payment phase ends, the client and server resume their interaction in the online sale. If the payment was successful, the server generates a receipt. Otherwise, the server generates an appropriate error notification. In case of a payment failure, the client can choose to abort the sale or retry the payment.

This example illustrates both aspects of web services identified in the Introduction. Online sale and online payment are examples of *sessions*—structured interactions involving multiple agents. On the other hand, the clients, servers and banks that participate in these sessions are examples of agents, each with its own control structure that determines how it evolves and moves from one session to another.

We propose to use finite-state automata to describe *session schemes* and *agents*. These prescribe the underlying structure from which concrete sessions are instantiated. Figure 1 depicts a session scheme for online sale, while Figure 2 shows a scheme for online payment. In these automata, transitions are labelled by shared actions, such as PayCardA and Authenticate. Each shared action is annotated with the names of the participants: for instance, C, S : Login indicates

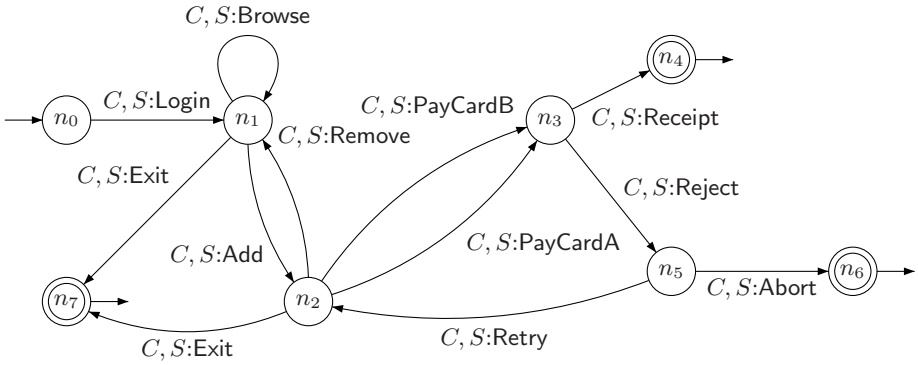

Fig. 1. Session template for Online Purchase

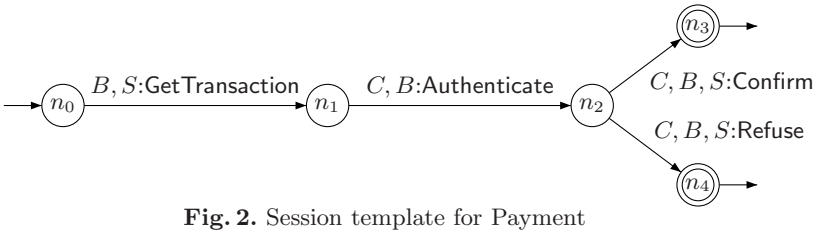

Fig. 2. Session template for Payment

that the action Login is shared by C and S. Here, C and S are not agents but abstract *roles*, to be played by actual agents when the scheme is instantiated as a concrete session.

Each session scheme has a start node, denoted by an incoming arrow and global final nodes marked by an outgoing arrow. Nodes with double circles are *return nodes* where one or more participants can exit the session without terminating the session itself. A session terminates when all participants have exited.

To ensure coordination between agents and across sessions, we need to equip the system with data. Each agent has a set of local variables that are updated as it evolves. In addition, each concrete session has variables to indicate its state, including the identities of the agents playing the various roles defined in the underlying session scheme. We will allow transitions to be guarded, so that a shared action may be enabled or disabled depending on the current state of the participating agents.

In a session system, several sessions that are active simultaneously may share agents—for instance, a customer may participate in online sales with two distinct retailers with the same bank. Sessions sharing an agent share the variables of this agent, so one has to take care to avoid unwanted interferences across sessions. For example, if payment information pertaining to different sessions gets mixed, an authorization for a low-cost transaction may be misused to complete a high-cost purchase beyond the customer's credit limit. As we shall see, our model allows us to enforce controlled access to critical sections through variables and guards, and also verify that mutual exclusion is achieved through formal analysis of the model. For the sake of readability, we have not represented variables and

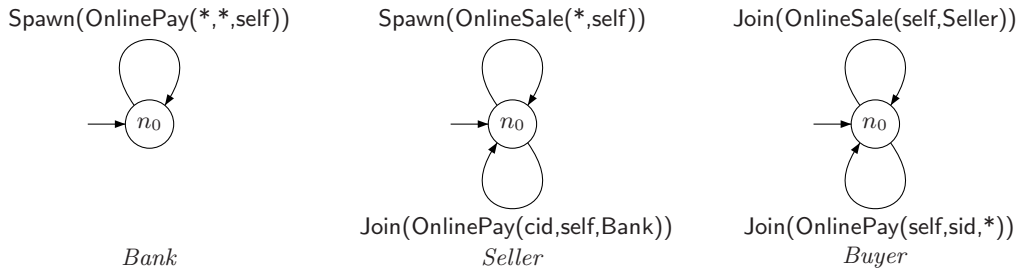

Fig. 3. Agents for Banks, Sellers and Buyers

guards in the examples here. Details can be found in the extended version of this work [5].

The other half of the system description consists of specifications for the agents. The agents *Bank*, *Seller* and *Client* are shown in Figure 3. There is one automaton for each agent: in this example, each agent has only a single state.

The typical actions of an agent are to spawn a new instance of a session scheme and to join an existing session. In this example, OnlinePay sessions are spawned by the bank and joined by the buyer and seller while OnlineSale sessions are spawned by the seller and joined by the buyer. The actions Spawn and Join refer to a session scheme with parameters that denote the association of agents to roles. For instance, the bank's action Spawn(OnlinePay(*,*,self)) spawns a new instance of the session scheme OnlinePay in which the current bank agent, self, plays the third role and the other two roles are left open for arbitrary agents. On the other hand, the buyer's action Join(OnlineSale(self,Seller)) says that the agent is willing to join any existing OnlineSale session in which the other participant is an instance of Seller, while Join(OnlinePay(self,sid,*)) says that the agent wants to join an OnlinePay session with a specific seller agent sid in the second role, but without any constraint on the bank playing the third role.

3 Session systems

A *session system* has a finite set of *agents* identified by names. Each agent has a finite data store and a finite repository of links to other agents. Agents operate at two levels. Individually, an agent executes a sequence of commands that determine its interactions with the other agents. Collectively, interactions are grouped into sessions. Within sessions, sets of agents perform synchronized actions, updating their respective data stores and link repositories.

An agent can *spawn* sessions from predefined session schemes or *join* existing sessions. It can also *kill* sessions and *quit* them. Each agent has a set of local variables—the state of an agent is given by the current values of these variables.

Session schemes provide templates for interaction patterns involving an abstract set of roles. A *session* is an instance of such a scheme in which concrete agents are associated with the abstract roles. A session progresses through the execution of synchronized actions involving subsets of the participating agents. These actions are enabled through guards that depend on the identities and states of the participating agents.

There may be multiple instances of a given session scheme running at a given time. Agents cannot "name" or "address" individual sessions. However,

agents can supply constraints when creating or joining sessions to filter out sessions from the collection of active sessions. Agents can join existing sessions synchronously or asynchronously. Agents that join a session synchronously are normally released just before the session *dies*.

Each session has a set of *role* variables that are used to describe the current mapping of abstract roles to concrete agents as well as to record constraints on the identity and type of agents that may join in the future to play roles that are currently unassociated.

In addition to sessions and agents, our model presupposes a global *scheduler* that manages sessions and serves requests for joining sessions. Agents can *query* this scheduler for the presence of a session of some kind. Queries are answered only if such session exists in the system.

3.1 Preliminaries

Let $\mathcal{A}$ denote a fixed, finite set of agents and $\mathbb{B} = \{\mathsf{tt}, \mathsf{ff}\}$ denote the set of boolean values. We assume the existence of two distinguished values $\perp$ and $\top$, whose interpretation will be explained later.

Each agent manipulates a set of local variables, organized as follows. There is a fixed set $X = X_A \uplus X_B$ of variable names, where X_A is the set of *agent variables*, including the distinguished variable self, and variables in X_B are boolean. A valuation of X is a pair of maps $V = (V_A, V_B)$ where $V_A : X_A \to \mathcal{A} \cup \{\perp\}$ and $V_B : X_B \to \mathbb{B} \cup \{\perp\}$. The variable self is a fixed read-only value: for agent a, $V_A(\mathit{self})$ always evaluates to a.

Each agent has a local copy of the set X. For $a \in \mathcal{A}$ and $x \in X$, $a.x$ denotes agent a's local copy of x. Though variables are local to agents, shared actions can observe and update local variables of all participating agents. When referring to variables and valuations of multiple agents simultaneously, we write $X^a = X^a_A \uplus X^a_B$ and $V^a = (V^a_A, V^a_B)$ to refer to the local variables and valuation of agent a, respectively.

In addition, session schemes are provided with a finite set Y of *role variables*, including a distinguished variable owner. Variables in Y are used to keep track of agents joining a session. A valuation of Y is a map $W : Y \to \mathcal{A} \cup \{\perp, \top\}$. The value $\top$ indicates that a role has been completed, so the corresponding agent is released from the session. When $W(y)$ is defined, we write $y.x$ as an abbreviation for $W(y).x$, the local copy of variable x in agent $W(y)$. In addition, we also equip each session with a *constraint map* $C : Y \to 2^{\mathcal{A}}$ that specifies constraints on the agents that can play each role. The set $C(y)$ indicates the set of agents that is compatible with the role y. We interpret $C(y) = \emptyset$ as an unconstrained role, rather than as a role that is impossible to fulfil.

3.2 Session schemes

A session scheme is a finite automaton with guarded transitions labelled by shared actions. Formally, a *session scheme* over a set of role variables Y is a tuple $S = (N, n_0, \Sigma, \ell, \delta)$, where:

- N is a finite set of *session nodes*, with an initial node n_0.

- Σ is a finite alphabet of actions that includes the special action Die that prematurely kills a session.
- $\ell : \Sigma \times Y \to \{\bot, +, \top\}$ defines for each shared action $\sigma \in \Sigma$ and role $y \in Y$ the participation of y in σ.
  - If $\ell(\sigma, y) = \bot$, y is not involved in σ: σ can execute even if $W(y) = \bot$.
  - If $\ell(\sigma, y) \neq \bot$, y is involved in σ: we must have $W(y) \neq \bot$ for σ to occur.
  - If $\ell(\sigma, y) = \top$, y terminates with action σ, and then agent $W(y)$ is released if it joined the session synchronously.
- $\delta \subseteq N \times G \times \Sigma \times U \times N$, is a *transition relation* between nodes, where G is the set of *guards*, and U is the set of *update functions*.

A transition (n, g, σ, u, n') means that a session can move from node n to node n' when guard g holds with respect to W, the current valuations of Y and $\{V^a\}_{a \in \mathcal{A}}$, the current valuations of all the agents in the system. These valuations are then updated as specified by u. The guard g and update u can only read and modify values of variables for roles y such that $\ell(\sigma, y) \neq \bot$. A guard g is a boolean combination of assertions of the form $y.x_1$ and $y_1 = y_2.x_2$. The literal $y.x_1$ is true if $W(y) = a \in \mathcal{A}$ and $V_B^a(x_1) = \mathsf{tt}$. The literal $y_1 = y_2.x_2$ is true if $W(y_1) \neq \top$, $W(y_2) = a \in \mathcal{A}$ and $W(y_1) = V_A^a(x_2)$. We lift this in the usual way to define the truth of the guard g.

3.3 Sessions

A session is an instance of a session template with roles assigned to agents in $\mathcal{A}$. Not all roles need to be defined in order for a session to be active—an action σ can be performed provided $W(y)$ is defined for every role that takes part in σ.

The constraint map C controls which agents can join the session in as yet undefined roles, as we shall see later.

We associate with each session a partial *return map* ρ from roles to states of agents. If $\rho(y)$ is defined, it means that the agent $W(y)$ is blocked and waiting for the session to end. Whenever $W(y)$ terminates in this session, or the session executes the action Die or it is killed by another agent, $W(y)$ resumes in the state $\rho(y)$.

3.4 Agents

The behaviour of an agent a is described by a tuple (Q, E, Δ, q_0) where

- Q is the set of control states, with initial state q_0.
- $\Delta \subseteq Q \times G \times E \times Q$ is the transition relation, where G is the set of *guards* over X^a. For simplicity, we define a guard as any function that maps each valuation $V^a = (V_A^a, V_B^a)$ of a to either tt or ff.
- E is a set of labels defining the *effect* of the transition, as described below.

Variable assignment $x := e$, where $x \in X$ and e is an expression over $\mathcal{A} \cup \mathbb{B} \cup \{\bot\} \cup X$ that is compatible with the type of x.

Asynchronous session creation $ASpawn(s, l)$, where s is a session scheme, and l is a list of constraints of the form $y = x$ where $y \in Y$ and $x \in X_A$. The variables *self* and *owner* should not appear in the constraints. $ASpawn(s, l)$ does not execute if $V(x) = \perp$ for some variable x occurring in the constraints. The new session is created with a valuation W such that $W(\mathit{owner}) = V(\mathit{self})$ and $W(y) = \perp$ for every other $y \in Y$. We also define the constraint map for the session as follows: $V(x) \in C(y)$ if and only if the constraint $y = x$ is in l.

Synchronous session creation $SSpawn(s, l)$, like asynchronous session creation, with the difference that the agent gives up control. This action sets the return map $\rho(\mathit{owner})$ to the target state of the transition carrying the spawn instruction to indicate where control returns when this agent's role terminates, when the session dies, or when the session is killed.

Asynchronous join $AJoin(s, y, l)$, where the variable y is the role of session scheme s that the process takes on joining the session and l specifies constraints of the form $y' = x'$, with $y' \in Y$ and $x' \in X_A$. $AJoin(s, y, l)$ does not execute if $V(x') = \perp$ for any variable x' occurring in the constraints.
Otherwise, it produces a *pending join request* (a, s, y, ϕ) where $a = V(\mathit{self})$. The map $\phi : Y \to 2^{\mathcal{A}}$ serves to filter out sessions from the collection of running sessions and is defined by $V(x') \in \phi(y')$ if and only if the constraint $y' = x'$ appears in l.
The join request $AJoin(s, y, l)$ is granted with respect to a session of type s with valuation W and constraint C if $W(y) = \perp$ and $V(\mathit{self}) \in C(y)$ and also, for each y', $W(y') \in \phi(y')$. Pending requests are dealt with asynchonously: that is, control returns to the agent immediately, without waiting for the join request to be granted.

Plain join $PJoin(s, y, l)$ is like asynchronous join, except that this command can be executed only if and when a session of type s meeting contraint l exists. Thus, after the command, the agent has already a role in the joined session and proceeds in the target state of the transition with the join instruction.

Synchronous join $SJoin(s, y, l)$, like plain join, except that control returns to the agent only after the session that it joins ends: that is, this agent's role terminates, the session dies or the session is killed. This action sets the return map $\rho(y)$ to the target state of the transition carrying the join instruction to indicate where control returns.

Query $Query(s, l)$, where list l specifies constraints of the form $x = y$ for $x \in X_A$ and $y \in Y$ (the variables *self* and *owner* may appear in these constraints). $Query(s, l)$ may execute even though $V(x) = \perp$ for some variable x occurring in the constraints. This command executes in an atomic step when some session with scheme s and valuation W satisfies all constraints $x = y$: that is, for every $x \in X_A$, if $V(x) \neq \perp$ then $V(x) = W(y)$ and if $V(x) = \perp$ then $W(y) \notin \{\perp, \top\}$. If the query succeeds, for each constraint $x = y$, $V(x)$ is updated to $W(y)$. In particular, if $V(x)$ was earlier $\perp$, x now acquires the value $W(y)$.

Kill *Kill*, kills all sessions created by the agent $V(\mathit{self})$. This has the same effect as when these sessions execute the action Die.

Quit *Quit*, agent $V(\mathsf{self})$ leaves all sessions that it has entered. This has no effect other than removing this agent from all session environments.

A major difference between creating and joining a session is that the creator of a session owns the session and can kill it, whereas an agent that has joined a session can only quit, in which case the session stays active if some roles have not yet terminated.

Joining a session asynchronously is like thread creation: the agent that makes a join request does not have to wait for the completion of the activities resulting from the firing of the join transition.

On the other hand, joining a session synchronously is like a remote procedure call from the perspective of the joining agent since it loses control. From the perspective of the joined session, no new incarnation of a session scheme is produced. The calling agent recovers control when the session it has joined terminates—the return state of the calling agent is kept track of by the joined session in the return map ρ.

Let us now comment about plain joins, which are in between asynchronous and synchronous joins. Like a synchronous join, a plain join $PJoin(s, y, l')$ cannot be executed before there exists some session of the specified scheme s with role y free. Like an asynchronous join, an agent that executes a plain join does not have to wait for the completion of the activities resulting from the firing of the join transition. Thus, for an agent a, using a transition $PJoin(s, y, l')$ amounts to waiting for another agent a' to spawn a new session, if needed, before a moves to another state. With the three forms of joins, one can easily model synchronization mechanisms, remote procedure calls, or threading mechanisms.

Finally, we comment on the difference between quit and kill. An agent that *Quits* leaves all sessions it has entered—that is, it stops playing a role in each of them, but does not otherwise affect the continuation of these sessions for those agents still engaged in active roles. This is essential to model collaborative frameworks in which the number of participants is not fixed in advance—for example, consider chat sessions where participants can join and leave freely. Conversely, *Kill* allows the owner of a session to close it unilaterally, hence stopping the service (but without killing the participants ...)—note that, to ensure robustness, only the owner of a session can kill it.

3.5 Scheduling

The effect of agents' actions on sessions and session actions are handled by the scheduler. We do not give details about how this scheduler is implemented—for instance, it could be via a shared memory manager. We assume that the scheduler keeps track of all active sessions and pending session requests. Serving a session request just consists of finding a running session of type s whose valuation W is compatible with the constraint ϕ of a session demand $sd = (a, s, y, \phi)$ and assigning role y to agent a in this running session. We denote this by a specific action labelled *Serve*.

4 Semantics of session systems

Session systems and configurations

Let $\mathcal{A}$ be a set of agents, X a set of variables, Y a set of role variables and $\mathcal{S}$ a set of sessions defined over X and Y. The tuple $(\mathcal{A}, \mathcal{S}, X, Y)$ defines a *session system*. A *session configuration* is a tuple (s, n, W, C, ρ), where s is a session scheme name, n is a state of session scheme s, W is a valuation of Y, C is a constraint on roles and ρ is a return map.

An *agent configuration* for an agent $a \in \mathcal{A}$ is a pair (q, V) where q is a state of the agent, and V is a valuation for variables in X. A *session system configuration* is a triple (Ψ, Γ, P), where Ψ associates a configuration to each agent $a \in A$, Γ is a set of session configurations, and P is a set of pending demands to join sessions. The following proposition ensures that configurations can be represented as finitely indexed multisets, and encoded as vectors, or as markings of a Petri net.

Proposition 1. *Let $\mathcal{A}$ be a finite set of agents and $\mathcal{S}$ be a set of session schemes over finite sets of variables X and Y. If $\mathcal{A}$ and $\mathcal{S}$ are defined over finite sets of states Q_P and Q_S, respectively, then the set $\mathcal{C}$ of session systems configurations that are definable over $\mathcal{A}, \mathcal{S}, X, Y$ is isomorphic to $Q_P^{|\mathcal{A}|} \times 2^{|X_B|\cdot|\mathcal{A}|} \times \mathcal{A}^{|X_A|\cdot|\mathcal{A}|} \times \mathbb{N}^K$, where $K = |\mathcal{S}| \cdot |Q_S| \cdot |Y|^{2|\mathcal{A}|+|Q_P|+2} + |\mathcal{A}| \cdot |\mathcal{S}| \cdot |Y|^{|\mathcal{A}|+1}$*

A session system moves from one configuration to another by performing an action. The obvious actions are process moves from E (spawning a session, joining a session, query, kill, quit) and session moves from Σ (shared actions, including the special action Die). In addition, we have internal system moves that *serve* requests to join a session. We say that a configuration χ' is a *successor* of a configuration χ via action $\sigma \in E \cup \Sigma \cup \{Serve\}$, and write $\chi \xrightarrow{\sigma} \chi'$, if and only if starting from χ, the effect of applying σ produces configuration χ'.

Reset Post-G nets

A *(labelled) Petri net* is a structure (P, T, λ, m_0, F) where P is a set of places, T is a set of transitions, λ is a function that associates a label to each transition of T, $m_0 : p \to \mathbb{N}$ associates a non-negative integer to each place of P, and $F : (P \times T) \cup (T \times P) \to \mathbb{N}$ is a weighted flow relation. A *marking* $m : P \to \mathbb{N}$ distributes tokens across the places. A transition t is enabled at m if each place p has at least $F(p, t)$ tokens. When t fires, the marking m is transformed to a new marking m' such that $m'(p) = m(p) - F(p, t) + F(t, p)$ for every place p.

In a *generalized self-modifying net* (G-net), the flow relation is enhanced to be of the form $F : (P \times T) \cup (T \times P) \to \mathbb{N}[P]$. In other words, the weights on the edges between places and transitions are polynomials over the contents of places in P. These polynomials are evaluated relative to the current marking to determine whether a transition is enabled and compute the effect of firing it.

In *Reset Post-G nets*, the input polynomials $F(p, t)$ are restricted so that $F(p, t) = \{p\}$ or $F(p, t) \in \mathbb{N}$. The term *reset* refers to the fact that every edge from a place p to a transition t weighted by a marking-dependent polynomial

is in fact weighted by the monomial p, which corresponds to resetting place p. Reset Post-G nets are a very expressive class of Petri Nets, but yet several key properties of nets such as termination and coverability remain decidable for this class [6]. In the rest of the paper, we will only consider Reset Post-G nets such that $F(p,t) = p$ or $F(p,t) \in \{0,1\}$, and such that $F(t,p) \in \{0,1\}$ or $F(t,p)$ is a sum of places p' such that $F(p',t) = p'$.

Claim. Let $(\mathcal{A}, \mathcal{S}, X, Y)$ be a session system starting in a configuration χ_0. Then the transition system $(\mathcal{C}, \chi_0, \longrightarrow)$ is the marking graph of a Reset Post-G net.

We establish this claim by building a Reset Post-G net whose marking graph is isomorphic to the set of configurations of the session system, and whose transitions encode moves from one configuration to another. From $(\mathcal{A}, \mathcal{S}, X, Y)$, we build the following subsets of places:

- $P_{Q,\mathcal{A}} = \{p_{q,a}, \ldots\}$ associates a place to each pair (q,a) where $a \in \mathcal{A}$ is an agent and q is a state of a. Since the set of states Q_P of all agents is finite, the set $P_{Q,\mathcal{A}}$ is finite as well.
- $P_{V,\mathcal{A}} = \{p_{v,a}, \ldots\}$ associates a place to each pair (v,a) where $a \in \mathcal{A}$ is an agent and v is a valuation of X. Since X is finite and the variables in X range over finite domains, $P_{V,\mathcal{A}}$ is a finite set.
- $P_{SC} = \{p_{sc}, \ldots\}$ is a set of places indexed by session configurations—that is, there exists a place p_{sc} for every tuple $sc = (s, n, W, C, \rho)$ that describes a valid session configuration.
- Finally, $P_D = \{p_{sd}, \ldots\}$ is a set of places indexed by join requests—that is, we have one place for every tuple $sd = (a, s, y, \phi)$ representing a join action.

We can now define the transitions of the Reset Post-G net. As discussed earlier, each action that transforms a session configuration is either a process move, or a session move, or an internal system move that serves a pending join request. Each move of the session system is represented by a finite set of net transitions. This representation is not a bijection, because, for instance, a move of an agent can be enabled in more than one valuation and in more than one environment.

For each agent $a = (Q, E, \Delta, q_0)$, and transition $t = (q, g, e, q')$ in Δ, we build net transitions and flow relations as follows:

- When $t = (q, g, e, q')$ is a variable assignment, for every valuation v satisfying the guard g, we construct a transition $t_{e,v}$ such that $\lambda(t_{e,v}) = e$, with preset $\{p_{v,a}, p_{q,a}\}$, postset $\{p_{v',a}, p_{q',a}\}$, and flow relations $F(p_{v,a}, t) = F(p_{q,a}, t) = F(t, p_{v',a}) = F(t, p_{q',a}) = 1$, where $v' = e(v)$ is the result of applying e to v.
- When $t = (q, g, e, q')$ with $e = \mathit{ASpawn}(s, l)$, we construct one transition $t_{v,e}$ for each valuation v that satisfies the guard g, with preset $\{p_{v,a}, p_{q,a}\}$, postset $\{p_{v,a}, p_{q',a}, p_{sc}\}$, letting $sc = (s, n_0, W, C, \rho_\emptyset)$ where n_0 is the initial state of s, $W(\mathsf{owner}) = a$ and $W(y) = \perp$ for all other roles y, C is generated by l and $\rho_\emptyset$ is the empty map. As for assignment transitions, we let $F(p_{v,a}, t) = F(p_{q,a}, t) = F(t, p_{v,a}) = F(t, p_{q',a}) = F(t, p_{sc}) = 1$.

- When $t = (q, g, e, q')$ with $e = SSpawn(s, l)$, we construct one transition $t_{v,e}$ for each valuation v that satisfies the guard g, with preset $\{p_{v,a}, p_{q,a}\}$ and postset $\{p_{v,a}, p_{sc}\}$, letting $sc = (s, n_0, W, C, \rho)$ where n_0 is the initial state of s, $W(\mathsf{owner}) = a$ and $W(y) = \perp$ for all other roles y, C is generated by l and $\rho(\mathsf{owner}) = q'$. We also let $F(p_{v,a}, t) = F(p_{q,a}, t) = F(t, p_{v,a}) = F(t, p_{sc}) = 1$. Note that with synchronous session creation, agent a loses control, and will resume in state q' after its role in s terminates. This information is kept in the return map ρ in sc.
- When $t = (q, g, e, q')$ with $e = AJoin(s, y, l)$, we construct a transition $t_{v,e}$ labelled by e for each valuation v that satisfies the guard g, with preset $\{p_{v,a}, p_{q,a}\}$ and postset $\{p_{v,a}, p_{q',a}, p_{sd}\}$, where $sd = (a, s, y, \phi)$ with map ϕ derived from the constraints in l. The flow relation is given by $F(p_{v,a}, t) = F(p_{q,a}, t) = F(t, p_{v,a}) = F(t, p_{q',a}) = F(t, p_{sd}) = 1$.
- When $t = (q, g, e, q')$ with $e = SJoin(s, y, l')$, we construct a transition $t_{v,e,sc}$ labelled by e for every valuation v that satisfies the guard g and for every session configuration $sc = (s, n, W, C, \rho)$ meeting constraint l' such that $W(y) = \perp$ and $a \in C(y)$. The preset of each transition is $\{p_{v,a}, p_{q,a}, p_{sc}\}$ and the postset is $\{p_{v,a}, p_{sc'}\}$, where $sc' = (s, n, W', C', \rho')$ is an updated session configuration in which $W'(y) = a$, $\rho'(y) = q'$, and C' is obtained by adding to C the constraints in l'. The flow relation is given by $F(p_{v,a}, t) = F(p_{q,a}, t) = F(p_{sc}, t) = F(t, p_{v,a}) = F(t, p_{sc'}) = 1$.
- When $t = (q, g, e, q')$ with $e = PJoin(s, y, l')$, we construct a transition $t_{v,e,sc}$ labelled by e for every valuation v satisfying the guard g and for every session configuration $sc = (s, n, W, C, \rho)$ meeting constraint l' such that $W(y) = \perp$ and $a \in C(y)$. The preset of each transition is $\{p_{v,a}, p_{q,a}, p_{sc}\}$ and the postset is $\{p_{v,a}, p_{q',a}, p_{sc'}\}$, where $sc' = (s, n, W', C', \rho)$ is an updated session configuration in which $W'(y) = a$ and C' is obtained by adding to C the constraints in l'. Unlike with synchronous join, the return map ρ is unchanged in sc' and control is returned via the output place $p_{q',a}$ to the agent executing the join instruction. The flow relation is given by $F(p_{v,a}, t) = F(p_{q,a}, t) = F(p_{sc}, t) = F(t, p_{v,a}) = F(t, p_{q',a}) = F(t, p_{sc'}) = 1$.
- When $t = (q, g, e, q')$ with $e = Query(s, l)$, we construct a transition $t_{v,e,v'}$ labelled by e for every valuation v satisfying the guard g, for every session configuration $sc = (s, n, W, C, \rho)$ meeting constraint l, and for every valuation v' computed from v by adding the bindings of agents induced by the constraints in l and the bindings of agents in W. The preset of each transition is $\{p_{v,a}, p_{q,a}, p_{sc}\}$ and the postset is $\{p_{v',a}, p_{q',a}, p_{sc}\}$. The flow relation is given by $F(p_{v,a}, t) = F(p_{q,a}, t) = F(p_{sc}, t) = F(t, p_{v',a}) = F(t, p_{q',a}) = F(t, p_{sc}) = 1$. Note that the transition does not consume any token from the place p_{sc}—it only tests the presence of a token.
- When $t = (q, g, e, q')$ with $e = Kill$, we construct a transition $t_{v,e}$ for every valuation v satisfying the guard g. Each of these transitions has as preset $\{p_{v,a}, p_{q,a}\} \cup \{p_{sc_i} \mid sc_i = (s, n, W, C, \rho) \wedge W(owner) = a\}$. The postset of the transition is the union of $\{p_{v,a}, p_{q',a}\}$ and the set of all places p_{q_j,b_j} such that b_j is a process which has issued a Synhronous Join with return state q_j.

The flow relation is defined as follows: $F(p_{v,a}, t) = F(p_{q,a}, t) = F(t, p_{v,a}) = F(t, p_{q',a}) = 1$, $F(p_{sc_i}, t) = p_{sc_i}$ for every session configuration sc_i owned by agent a (the transition consumes all sessions created by a), and $F(t, p_{q_j,b_j}) = \sum \{p_{sc} \mid sc = (s', n', W', C', \rho') \wedge W'(y) = b_j \wedge \rho'(y) = q_j\}$. Note that an agent b_j can be blocked in at most one session, so $F(t, p_{q_j,b_j}) \leq 1$ and control is returned to agent b_j only when it was blocked.

– When $t = (q, g, e, q')$ with $e = \mathit{Quit}$, we construct a transition $t_{v,e}$ for every valuation v satisfying the guard g. Each of these transitions has preset $\{p_{v,a}, p_{q,a}\} \cup NT$, where $NT = \{p_{sc_i} \mid sc_i = (s, n, W, C, \rho) \wedge (\exists y)(W(y) = a)\}$. The postset of the transition is $\{p_{v,a}, p_{q',a}\} \cup NT'$, where NT' is the set of places $p_{sc'_i}$ representing session configurations $sc'_i = (s, n, W', C, \rho)$ such that, for some $sc_i = (s, n, W, C, \rho)$ in NT, $W(y) = a \Rightarrow W'(y) = \top$ and $W'(y) = W(y)$ otherwise for all y, and $W'(y) \neq \top$ for some y. The flow relation is defined as follows: $F(p_{v,a}, t) = F(p_{q,a}, t) = F(t, p_{v,a}) = F(t, p_{q',a}) = 1$, $F(p_{sc_i}, t) = p_{sc_i}$ for every session configuration sc_i appearing in NT (the transition consumes all sessions involving a), and $F(t, p_{sc'_i}) = p_{sc_i}$ for every session configuration sc'_i in NT'. This way, session configurations involving a, but still having other non-terminated roles, are transformed into session configurations without agent a.

The second part of the translation concerns the internal progress of sessions. We will distinguish two kinds of transitions, depending on whether the translated action brings the session to termination or not.

We consider first the non terminating actions $\sigma \neq \mathsf{Die}$. Let $sc = (s, n, W, C, \rho)$ be a session configuration, and let $(n, g, \sigma, u, n') \in \delta$ with guard g and update u. Executing such an action in sc is conditioned to the satisfaction of guard g w.r.t. W and transforms sc into a configuration $sc' = (s, n', W', C', \rho)$, where $W'(y) = \top$ if $\ell(\sigma, y) = \top$ and $W'(y) = W(y)$ otherwise. Then for every valuation v that satisfies g, we construct a transistion $t_{sc,v}$ with preset $\{p_{v,a}, p_{sc}\}$ and postset $\{p_{sc'}, p_{v',a}\} \cup P_{\rho,\sigma}$, where $P_{\rho,\sigma} = \{p_{q'',a} \mid \exists y, W(y) \neq W'(y) \wedge \rho(y) = q'' \wedge W(y) = a\}$ and $v' = u(v)$ is the result of applying u to v. In other words, all agents that have joined the session synchronously and that leave the session by action σ resume their activity in the control state q'' specified at join time. The flow relation is $F(p_{v,a}, t_{sc,v}) = F(p_{sc}, t_{sc,v}) = F(t_{sc,v}, p_{sc'}) = F(t_{sc,v}, p_{v',a}) = 1$. Moreover, for every place $p_{q'',a}$ in $P_{\rho,\sigma}$, we let $F(t_{sc,v}, p_{q'',a}) = 1$.

We now consider the terminating action Die. For every session configuration $sc = (s, n, W, C, \rho)$, and for every $(n, g, \sigma, u, n') \in \delta$ such that $\sigma = \mathsf{Die}$ and guard g is satisfied w.r.t. W, we construt a transition $t_{sc,die}$ labeled by action Die with the preset p_{sc} and the postset $P_\rho = \{p_{q'',a} \mid \exists y, \rho(y) = q'' \wedge W(y) = a\}$. The flow relation is $F(p_{sc}, t_{sc,die}) = 1$, and $F(t_{sc,die}, p) = 1$ for every p in P_ρ.

Finally, we have to translate into net transitions the system moves that serve pending join requests. Serving a request just consists of removing the request from the set of pending requests and modifying the configuration of a session compatible with this request in the set of session configurations.

For every Asynchronous Join pending demand $sd = (a, s, y, \phi)$ and for every session configuration $sc = (s, n, W, C, \rho)$ compatible with this request, we con-

struct a transition $t_{sc,sd}$ labeled with the internal action *Serve*. The preset of $t_{sc,sd}$ is $\{p_{sc}, p_{sd}\}$ and its postset is $\{p_{sc'}\}$, where $sc' = (s, n, W', C', \rho)$ is obtained from sc by setting $W'(y) = a$ (and $W'(y') = W(y')$ for every $y' \neq y$) and letting C' be C augmented by the constraints in ϕ. The flow relation is given by $F(p_{sc}, t_{sc,sd}) = F(p_{sd}, t_{sc,sd}) = F(t_{sc,sd}, p_{sc'}) = 1$.

Note that for every transition of the global Petri net obtained in the end, the weight of the flow relation from a place p to a transition t is either 0 or 1 or $F(p,t) = p$, whereas the weight of the flow relation from a transition t to a place p is either 0 or 1 or a polynomial over the contents of a set of places. Hence, the semantic model for session systems corresponds to Reset Post-G nets.

Definition 1. *Let $\chi = (\Psi, \Gamma, P)$ and $\chi' = (\Psi', \Gamma', P')$ be two configurations of a session system. Configuration χ' is* reachable *from χ if and only if there exists a sequence of moves starting from χ that leads to configuration χ'. Configuration χ'* covers *χ, denoted by $\chi \sqsubseteq \chi'$, iff $\Psi' = \Psi$, and for every session configuration sc and session demand sd we have $\Gamma[sc] \leq \Gamma'[sc]$ and $P[sd] \leq P'[sd]$. Configuration χ' is* coverable *from χ iff there exists a sequence of moves starting from χ and leading to a configuration χ'' such that $\chi' \sqsubset \chi''$. A session system is* bounded *iff there exists some constant B such that $\Gamma[sc] \leq B$ and $P[sd] \leq B$ for every sc and sd in any reachable configuration $\chi = (\Psi, \Gamma, P)$.*

Proposition 2. *Given a session system $(\mathcal{A}, \mathcal{S}, X, Y)$ with initial configuration χ_0 and a configuration $\chi \in \mathcal{C}$, one can decide whether one can reach a marking χ' from χ_0 that covers χ. Termination—that is, the absence of infinite runs starting from χ_0— is also decidable.*

Proof: This proposition stems directly from the properties of Reset Post-G Nets, for which coverability of a given configuration and termination are decidable. □

Coverability is an important issue for the kind of services we want to model. To illustrate this, let us return to the example of Section 2, where we wanted to check that an agent a does not participate in two payments at the same time. This property may be expressed as follows: "There is no reachable configuration of the system in which agent a participates as a customer in at least two purchase sessions s, s' whose states are included in $\{n_3, n_5\}$". This property reduces to a coverability check in the Reset Post-G net modelling the session system.

Proposition 3. *Given a session system $(\mathcal{A}, \mathcal{S}, X, Y)$ with initial configuration χ_0, one can decide neither upon the reachability of a given configuration $\chi \in \mathcal{C}$ from χ_0 nor on the boundedness of the session system.*

Proof Sketch: One can simulate Reset Petri Nets with session systems. Now, boundedness and exact reachability are undecidable for Reset Petri Nets. □

5 Conclusion

We have proposed a session-based formalism for modeling distributed orchestrations. We voluntarily limited the expressiveness of the language to ensure decidability of some important practical properties. Indeed, many properties of session systems, such as the possibility for an agent or for a session to perform a

given sequence of transitions, may be expressed as a coverability problem on Reset Post-G nets. As deadlock and exact reachability are undecidable in general, a natural question is how to restrict the model to enhance decidability.

A second issue to consider is the implementation of session systems. The natural implementation is a distributed architecture in which agents use only their local variables. However, agents share sessions that have to be managed globally, along with requests and queries. This means, in particular, that an implementation of session systems has to maintain a kind of shared memory that can be queried by agents. This can be costly, and a challenge is to provide implementations with the minimal synchronization.

A third issue is to consider session systems as descriptions of security protocols, and to see whether an environment can break security through legal use of the protocol. For instance, the well known session replay attack of the Needham-Schroeder protocol can apparently be modelled by a simple session type system, and the failure of the protocol (the existence of a session involving unexpected pairs of users) can be reduced to a coverability issue. Whether such an approach can be extended to more complex protocols for detecting unknown security failures is an open question.

References

1. S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: A data-centric perspective on web services. In *BDA02*, 2002.
2. S. Abiteboul, L. Segoufin, and V. Vianu. Static analysis of Active XML systems. In *PODS08*, pages 221–230, 2008.
3. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services (BPEL4WS). version 1.1, 2003.
4. R.M. Arlein, D. Dams, R.B. Hull, J.P. Letourneau, and K.S. Namjoshi. Telco meets the web: Programming shared-experience services. *Bell Labs Technical Journal*, 14(3):167–185, 2009.
5. P. Darondeau, L. Hlout, and M. Mukund. Assembling sessions (long version). Technical report, Inria Rennes and Chennai Mathematical Institute, 2011. `http://www.irisa.fr/distribcom/DST09/DHM-ATVA-Long.pdf`.
6. C. Dufourd, A. Finkel, and P. Schnoebelen. Reset nets between decidability and undecidability. In *Proc. of ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 103–115, 1998.
7. S. Hinz, K. Schmidt, and C. Stahl. Transforming bpel to petri nets. In *Business Process Management, 3rd International Conference, BPM 2005*, volume 3649 of *LNCS*, pages 220–235, 2005.
8. D. Kitchin, W.R. Cook, and J. Misra. A language for task orchestration and its semantic properties. In *CONCUR'06*, pages 477–491, 2006.
9. J. Misra and W. Cook. Computation orchestration. *Software and Systems Modeling*, 6(1):83–110, 2007.
10. W.M.P van der Aalst. The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
11. H.M.W. Verbeek and W.M.P van der Aalst. Analyzing BPEL processes using Petri nets. *Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, 8(1):59–78, 2005.