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, as they allow the implementation of decentralized transactions with multiple participants. Sessions enable the cooperation of workflows while at the same time avoiding the mixing of workflows from distinct transactions. Several languages such as BPEL, ORC, AXML have been proposed to implement Web Services. Sessions are usually implemented 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 in terms of coverability in Petri nets.

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. These interactions 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. Sessions are obtained as concrete instantiations of templates that define patterns of interaction between agents in terms of some "rules". A template can have multiple parallel instantiations as sessions, each with its own local "state".

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

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

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 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 [4] 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 post reset-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 There have been other frameworks proposed for defining 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 [5, 6], and declarative, rule-based formalisms such as AXML [1, 2]). Each of these approaches has its advantages and drawbacks.

A BPEL specification describes a set of independent communicating agents equipped with a rich set of control structures. Each agent is associated with a set of partners required to complete its activity and 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 that may be written in standard programming languages such as JAVA. 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 is purely declarative. An AXML description defines a set of rules for each site that provides and uses services. It deals very well with data since the model is based on rewriting of semi-structured documents described, for instance, in XML. However, the disadvantage of declarativeness is that AXML cannot make workflows explicit. AXML does not have a native notion of session either, and complex guarding mechanisms must be used to forbid transactions from mixing.

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. The model introduced in the next section can be translated to a class of Petri Nets. This is not the first time Petri Nets are used to model workflows in web services, as shown in [7–9].

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 abstractly 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.

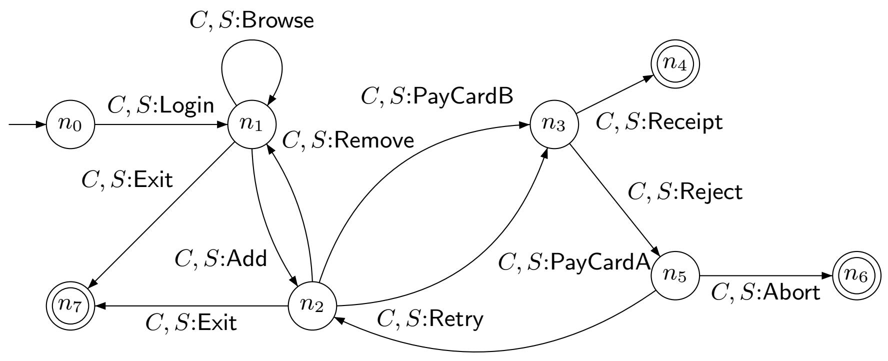

Fig. 1. Session template for Online Purchase

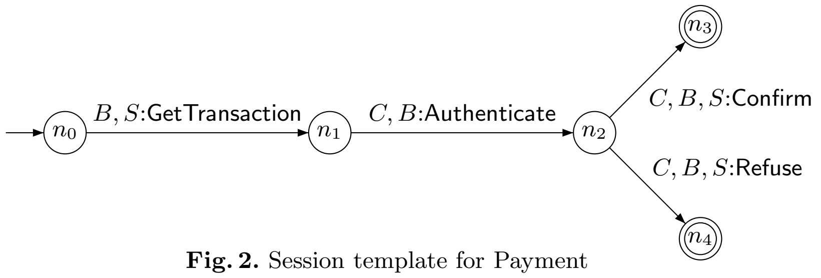

Fig. 2. Session template for Payment

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 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 will have a set of local variables that are updated as it evolves. In addition, each concrete session will have 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.

A typical example of the type of situation we have to take care of is the mixing of payment information across different online sale sessions for the same client. For instance, an authorization for a low-value transaction may be misused to complete a high-value purchase that is beyond the client's credit limit. To avoid this, we can use local variables to enforce that each client and server pair can be involved in at most one active online sale session at a time.

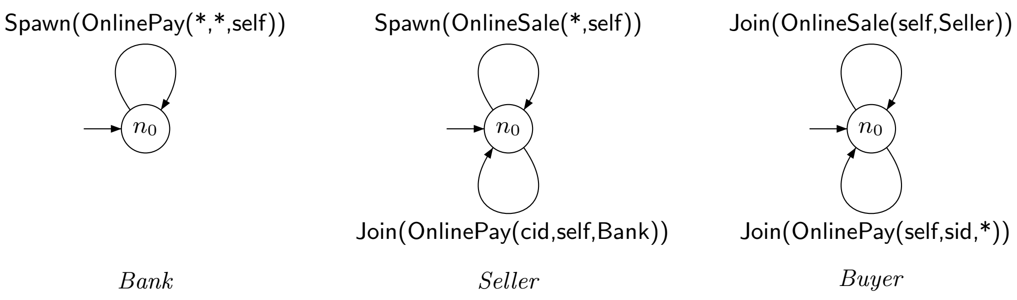
Spawn(OnlinePay(*,*,self))
n_0
Bank
Spawn(OnlineSale(*,self))
n_0
Join(OnlinePay(cid,self,Bank))
Seller
Join(OnlineSale(self,Seller))
n_0
Join(OnlinePay(self,sid,*))
Buyer

Fig. 3. Agents for Banks, Sellers and Buyers

Similarly, we can use auxiliary variables and guards to prevent undesirable situations such as entering a payment session without a preceding sale session.

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 agent playing the third role.

As we shall see later, it useful to have both asynchronous and synchronous versions of the Spawn and Join operations. We also need actions that model premature session termination. This situation could arise due to an unpredictable environment event, such as a network timeout, or a conscious decision by an agent to abort a session, such as a client deciding to cancel a payment.

In general, a session system may involve an arbitrary number of clients, servers and banks with some constraints enforced by variables and guards. For instance, each client can participate in an arbitrary number of sessions, but should only be able to perform a payment for one of them. While processing a payment to a server S, a client C can still browse the proposals of the same server in another session and fill its cart. A server can be involved in an arbitrary number of sessions, but can be in at most one payment session with each client.

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 *create* 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.

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.

Agents can join existing sessions synchronously or asynchronously. Agents that join a session synchronously are normally released just before the session *dies*. If the session dies prematurely, these agents are woken up with a warning, set via a special local variable.

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. For simplicity, we assume that these variables are organized as follows. There is a fixed set $X = X_A \uplus X_B$ of variable names. The set X_A is the set of *agent variables*, including the distinguished variable self. The variables in X_B are boolean and include the distinguished variable $\mathit{warning}$. 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 controlled by 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 $W : Y \rightarrow \mathcal{A} \cup \{\bot, \top\}$. 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 $y.x$ an 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* $C : Y \rightarrow 2^{\mathcal{A}}$ that specifies constraints on the agents that can play each role. The set $C(y)$ indicates the set of agents that is compatibile with the role y. We interpret $C(y) = \emptyset$ as an unconstrained role, rather than as a role with no compatible agents 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 and returns a warning to each synchronously participating agent, as described later.

– $\ell : \Sigma \times Y \rightarrow \{\bot, +, \top\}$ 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 this action. 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') and means that a session can move from a node n to a node n' when guard g holds for the current valuations W and $\{V^a\}_{a \in \mathcal{A}}$. These valuations are then updated as described by the update u.

  A guard g is a boolean combination of assertions of the form $y.x_1$ and $y_1 = y_2.x_2$. Let W be the current valuation of Y and $\{V_a\}_{a \in \mathcal{A}}$ be the current valuations of all the agents in the system. The literal $y.x_1$ is true if $W(y) = a \in \mathcal{A}$ and $V^a_B(y.x_1) = \mathsf{tt}$. 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.

  The guard g and update u can only read and modify values of variables for roles y such that $\ell(\sigma, y) \neq \bot$.

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 $\rho(y)$. In addition, if a session ends prematurely—that is, it dies or is killed—the special variable *warning* is set to tt.

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.
- $\Delta \subseteq Q \times G \times E \times T$ 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 $V^a = (V^a_A, V^a_B)$ 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 $\mathcal{A} \cup \mathbb{B} \cup \{\perp\} \cup X$ 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 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.

Blocking join $BJoin(s, y, l)$, like asynchronous join, except that with this command, the agent gives up control until the join request is granted. A blocking join is granted when a session of type s meeting contraint l exists. When control returns, the agent continues in the target state of the transition with the join instruction.

Synchronous join $SJoin(s, y, l)$, like blocking 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 spawn 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 $V(\mathit{self})$ leaves all sessions that it has entered. This has no effect other than removing this agent from all session environments.

In addition to sessions and agents, our model presupposes a global scheduler that manages sessions and serve requests for joining sessions. 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 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.

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.

Let χ, χ' be two configurations of a session systems. We say that χ' is a successor of χ via action $\sigma \in E \cup \Sigma \cup \{Serve\}$ 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 the 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 P and the firing rule is generalized so that these polynomials are evaluated to determine when a transition is enabled and the effect of firing the transition.

In a *reset Post-G net*, 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 the only non-trivial input weight to a transition corresponds to resetting a place.

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 $F(p',t) = p'$. Reset Post G nets are Petri Nets in which the contents of places can be reset (when $F(p,t) = p$), and in which outgoing flows fill places with a number of tokens that can be a polynomial over the contents of places in the net (seen as variables). This class is very expressive, but yet several properties (termination, coverabilty) remain decidable [4].

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 a session system, and whose transitions are Reset Post-G net transitions that encode moves from one configuration to another.

For a given session system $(\mathcal{A}, \mathcal{S}, X, Y)$, we build the following set of places:

- $P_{Q,\mathcal{A}} = \{p_{q,a}, \ldots\}$ that associates a place to each pair $a \in \mathcal{A}$ and state q of agent a. Since the set of states Q_P across all agents is finite, the set $P_{Q,\mathcal{A}}$ is finite as well.
- $P_{V,\mathcal{A}} = \{p_{v,a}, \ldots\}$ that associates a place to each $a \in \mathcal{A}$ and each valuation v of the variables X. Since 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 the set of possible 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, we have $P_D = \{p_{sd}, \ldots\}$, a set of places indexed by the possible join requests—that is, we have one place per tuple $sd = (a, s, y, \phi)$ generated by a join action.

We can now define the transitions of the net. As discussed earlier, each action that transforms a session configuration is either a process move, a session move, or an internal system move that serves a pending join request.

The mapping from session system moves to net transitions is not one to one. This is because a move of an agent can, for instance, be enabled in more than one valuation and address more that one kind of environment. However, this number will always be finite, as we shall show later.

Let a be an agent. We design a set $T_a = \bigcup_{t \in a} T_a^t$ by associating a set of net transitions T_a^t for each transition t of process a, as described below.

- Let $t = (q, e, q')$ be an assignment transition of process a. Then, for every valuation v satisfying the guard of e, we create a transition $t_{e,v}$ such that $\lambda(t_{e,v}) = e$, with preset $\{p_{v,a}, p_{q,a}\}$ and postset $\{p_{v',a}, p_{q',a}\}$, and flow relations $F(p_{v,a}, t) = F(p_{q,a}, t) = F(t, p_{v',a}) = F(t, p_{q',a}) = 1$.
- When t is an asynchronous session creation—that is, $t = (q, e, q')$ with $e = ASpawn(s, l)$—we create again one transition $t_{v,e}$ for each valuation v that satisfies the guard of the transition, with preset $\{p_{v,a}, p_{q,a}\}$ and postset $\{p_{v',a}, p_{q',a}, p_{sc}\}$, where $sc = (s, n, W, C, \rho_\emptyset)$ with $W(\mathsf{owner}) = a$ 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 have $F(p_{v,a}, t) = F(p_{q,a}, t) = F(t, p_{v',a}) = F(t, p_{q',a}) = F(t, p_{sc}) = 1$.

- When t is a synchronous session creation—that is, $t = (q, e, q')$ with $e = \mathit{SSpawn}(s, l)$—we again create one transition $t_{v,e}$ for each valuation v that satisfies the guard of the transition, with preset $\{p_{v,a}, p_{q,a}\}$ and postset $\{p_{v',a}, p_{sc}\}$, where $sc = (s, n, W, C, \rho)$ with $W(\mathsf{owner}) = a$ and $W(y) = \perp$ for all other roles y, C is generated by l and $\rho(\mathsf{owner}) = q'$. We also have $F(p_{v,a}, t) = F(p_{q,a}, t) = F(t, p_{v',a}) = F(t, p_{sc}) = 1$. Note that during synchronous session creation, agent a loses control, and will resume in state q' after its role terminates. This information is kept in the return map ρ.
- When t is an asynchronous join—that is, $t = (q, e, q')$ with $e = \mathit{AJoin}(s, y, l)$—we create a transition $t_{v,e}$ labelled by e for each valuation v that satisfies the guard of the transition, 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 ϕ generated by 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 is a synchronous join—that is, $t = (q, e, q')$ with $e = \mathit{SJoin}(s, y, l')$—we create a transition $t_{v,e,sc}$ labelled by e for every valuation v that satisfies the guard of the transition, and every session configuration $sc = (s, n, W, C, \rho)$ meeting constraint l'. 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 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 is a blocking join—that is, $t = (q, e, q')$ with $e = \mathit{BJoin}(s, y, l')$—we create a transition $t_{v,e,sc}$ labelled by e for every valuation v satisfying the guard of e, and every session configuration $sc = (s, n, W, C, \rho)$ meeting constraint l'. 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 configuration in which $W'(y) = a$, and C' is obtained by adding to C the constraints in l'. Compared to a synchronous join, note that the return map ρ is unchanged in sc', and control is returned to the agent executing the instruction via the output place $p_{q',a}$. 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 is a query—that is, $t = (q, e, q')$ with $e = \mathit{Query}(s, l)$—we create a transition $t_{v,e,v'}$ labelled by e for every valuation v satisfying guard of e, every session configuration $sc = (s, n, W, C, \rho)$ meeting constraint l, and every resulting valuation v' computed from v and the variable assignment in sc. 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 this kind of transition does not consume a token from the place p_{sc}—it only tests the presence of a token.
- When t is a kill—that is, $t = (q, e, q')$ with $e = \mathit{Kill}$—we create a transition $t_{v,e}$ for every valuation v satisfying guard of e. Each of these transitions has as preset $\{p_{v,a}, p_{q,a}\} \cup \{p_{sc_i} \mid sc_i = (s, n, W, C, \rho)\}$ for some s, n, W, C, ρ such that $W(\mathit{owner}) = a$. The postset of the transition is

$\{p_{v',a}, p_{q',a}\} \cup \{p_{q_j,b}\}$ such that q_j is the target state of some *SJoin* transition of process b. The flow relation is decomposed 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 owned by agent a, that is the transition consumes all sessions created by a, and $F(t, p_{q_j,b}) = \sum_{\rho_k(y)=q_j \wedge W(y)=b} p_{sc_k}$. Note that agent b can be blocked in at most one session, so we can be sure that $\sum p_{sc_k} \leq 1$, that is we return control to agent b only when it was blocked.

– When the considered transition t is labelled *Quit*, we create a transition $t_{v,e}$ for every valuation v satisfying the guard of e. Each of these transitions has preset $\{p_{v,a}, p_{q,a}\} \cup NT$, where $NT = \{p_{sc_i} \mid sc_i = (s, n, W, C, \rho)\}$ for some s, n, W, C, ρ where $W(y) = a$ for some role y. The postset of the transition is $\{p_{v',a}, p_{q',a}\} \cup NT'$, where $NT' = \{p_{sc'_i} \mid \exists sc_i \in NT\}$ such that each $sc'_i = (s, n, W', C, \rho)$ is obtained from $sc_i = (s, n, W, C, \rho)$ by setting $W'(y) = \top$ when $W(y) = a$. The flow relation is decomposed as follows: $F(p_{v,a}, t) = F(p_{q,a}, t) = F(t, p_{v',a}) = F(t, p_{q',a}) = 1$ and $F(p_{sc_i}, t) = p_{sc_i}$ for every session configuration appearing in the preset—that is, the transition consumes all sessions involving a, and $F(t, p_{sc'_i}) = p_{sc_i}$ for every sc'_i in NT'. This way, session configurations involving a, but with other remaining active roles, are transformed into session configurations without agent a.

The next part of this translation concerns the internal behavior of sessions. We will distinguish two kinds of transitions, depending on whether a session terminates after playing an action or not.

Let us first consider transitions associated with sessions that do not terminate; that is, the action is not 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 transforms a configuration 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 create a transistion $t_{sc,v}$ with preset $\{p_{v,a}, p_{sc}\}$ and postset $\{p_{sc'}, p_{v',a}\} \cup P_{\rho,\sigma}$, where $P_{\rho,\sigma} = \{p_{q'',a} \mid \exists y, W(y) \neq W'(y) \wedge \rho(y) = q'' \wedge W(y) = a\}$ with $v' = u(v)$. In other words, all agents that have joined the session synchronously and leave the session during action σ resume in the control state defined at call time. The flow relation is $F(p_{v,a}, t_{sc,v}) = F(p_{sc}, t_{sc,v}) = F(t_{sc,v}, p_{sc'}) = F(t_{sc,v}, p_{v',a}) = 1$ We also have, for every place $p_{q'',a}$ in $P_{\rho,\sigma}$, $F(t_{sc,v}, p_{q'',a}) = 1$.

For every session configuration $sc = (s, n, W, C, \rho)$ such that action Die can be executed from n, we create 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 system moves that serve pending join requests into net transitions. 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 session demand $sd = (a, s, y, \phi)$ and session configuration $sc = (s, n, W, C, \rho)$ compatible with the request, we create a transition $t_{sc,sd}$ labeled by 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 the session configuration obtained by updating configuration sc i.e. setting $W'(y) = a$ (and $W'(y') = W(y')$ for every $y' \neq y$) and C' is obtained from C by adding 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 this translation, the flow relation from the preset to the transition is either that of a standard Petri net or a reset transition—that is, $F(p, t) = p$. The flow relation from the transition to a place is either a standard Petri net flow relation—that is, $F(t, p) = 1$—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 2. *Let $\chi = (\Psi, \Gamma, P)$ be a configuration of a session system.*

- *We say that a configuration $\chi' = (\Psi', \Gamma', P')$ is* reachable *from configuration χ if and only if there exists a sequence of moves starting from χ that leads to configuration χ'.*
- *We say that χ'* covers *χ, and write $\chi \sqsubseteq \chi'$, iff $\Psi' = \Psi$, and for every session configuration sc and session demand sd we have $\Gamma[sc] \leq \Gamma'[sc]$ and $P[sd] \leq P'[sd]$.*
- *We say that a 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 in any reachable configuration $\chi = (\Psi, \Gamma, P)$ we have $\Gamma[sc] \leq B$ and $P[sd] \leq B$ for every sc and sd.*

Theorem 3. *Let $(\mathcal{A}, \mathcal{S}, X, Y)$ be a session system starting in a configuration χ_0. Then coverabiliy of some configuration $\chi \in \mathcal{C}$ is decidable. Termination—that is, absence of infinite runs starting from χ_0 is decidable.*

Proof: This theorem stems directly from the properties of reset post-G Nets, for which coverability of some configuration and termination are decidable. □

Theorem 4. *Let $(\mathcal{A}, \mathcal{S}, X, Y)$ be a session system starting in a configuration χ_0. Then reachability of some configuration $\chi \in \mathcal{C}$ from χ_0 and boundedness are undecidable problems.*

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

5 Conclusion

We have proposed a session-based formalism for modeling distributed orchestrations. We have voluntarily limited the expressiveness of the language to ensure decidability of some important practical properties of the model. 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. However, deadlock and exact reachability are undecidable in general. A natural question is how to restrict the model to enhance decidability.

A second issue is 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. Serge Abiteboul, Omar Benjelloun, Ioana Manolescu, Tova Milo, and Roger Weber. Active XML: A data-centric perspective on web services. In *BDA02*, 2002.
2. Serge Abiteboul, Luc Segoufin, and Victor Vianu. Static analysis of Active XML systems. pages 221–230, 2008.
3. Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language for web services (BPEL4WS). version 1.1, 2003.
4. Catherine Dufourd, Alain Finkel, and Ph. Schnoebelen. Reset nets between decidability and undecidability. In *Proc. of ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 103–115, 1998.
5. David Kitchin, William R. Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. In *CONCUR'06*, pages 477–491, 2006.
6. Jayadev Misra and William Cook. Computation orchestration. *Software and Systems Modeling*, 6(1):83–110, 2007.
7. Wil 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.
8. Wil M. P. van der Aalst and Kees van Hee. *Workflow management: Models, Methods, and Systems.* MIT Press, 2002.
9. H.M.W. Verbeek and Wil 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.