Sessions with an unbounded number of agents

S. Akshay
IIT Bombay
Email: akshayss@cse.iitb.ac.in

Loïc Hélouët
INRIA Rennes
Email: loic.helouet@inria.fr

Madhavan Mukund
Chennai Mathematical Institute
Email: madhavan@cmi.ac.in

***Abstract*—In web-based systems, agents engage in structured interactions called sessions. Sessions are logical units of computation, like transactions. However, unlike transactions, sessions cannot be isolated from each other. Thus, one has to verify that interference between sessions does not have unexpected side effects. A challenge in building a tractable model of sessions is that there is no a priori bound on the number of concurrently active agents and sessions in the system. Realistic specifications require agents to compare entities across sessions, but this must be modeled without assigning an unbounded set of unique identities to agents and sessions.**

We propose a model called session systems that allows an arbitrary number of concurrently active agents and sessions. Agents have a limited ability to remember partners across sessions. Configurations are represented as graphs and the operational semantics is described through graph-rewriting. Under reasonable restrictions, session systems turn out to be well-structured systems. This provides an effective verification algorithm for coverability properties. We show how to use this result to verify more elaborate business rules such as avoidance of conflicts of interest and the Chinese Wall Property.

***Keywords*-Services, WSTS, verification.**

I. Introduction

Web services involve many parties interacting with each other to achieve a goal. The communication between the participating agents typically follows a structured protocol and the entire sequence of interactions can be seen as a logical unit of computation, typically called a *session*.

Sessions exhibit a richer behaviour than conventional transactions. Transactions combine smaller steps into computational units that satisfy the ACID properties—atomicity, consistency, isolation and durability. In particular, each transaction is assumed to be independent. For transactions executing in parallel, the expected behaviour is specified in terms of notions such as serializability and linearizability that presuppose that transactions do not interact.

Sessions, on the other hand, typically need to interact to achieve the task at hand. Consider a scenario where a customer makes an online purchase using a credit card. There are three interactions: the customer interacts with the merchant to order the item, the merchant interacts with the bank to confirm the payment and the customer interacts with the bank to authenticate the payment. Logically, each is a separate session. However, the merchant cannot confirm the order before the two sessions with the bank are completed. Likewise, the customer authenticates the payment after the merchant tells the bank how much is to be paid and before the bank confirms the payment to the merchant.

To capture these features, we need a model that does more than just encapsulating a sequence of activities as an atomic block. The model must permit controlled interactions such as the ones described above, while taking care to disallow undesirable interference. For instance, authentication for one payment should not be reused for another purchase.

Our goal is to build a tractable model of sessions that is amenable to formal verification. In addition to allowing sessions to interact in a controlled manner, there is another challenge. An unbounded number of agents of a given type may be active simultaneously—e.g., customers at an online store. This allows unboundedly many sessions to be active in parallel. When sessions interact, we need to compare entities across sessions and maintaining unbounded sets of agent identities usually makes verification untractable.

Our first contribution is to extend the *session system* formalism proposed in [1] to model systems with an arbitrary number of active agents and sessions. We represent configurations of session systems as labelled graphs. The semantics is given in terms of finite graph rewriting rules.

Our second contribution is to propose effective verification techniques for session systems. Not surprisingly, reachability is undecidable for unbounded session systems. However, in many cases, the weaker property of *coverability*—whether a configuration embeds a given pattern—suffices. We show that, under reasonable restrictions, session systems fall within a class of well-structured transition systems (WSTS) for which coverability is decidable. We then use the decidability of coverability to verify properties expressing "business rules", such as avoidance of *conflict of interest* and the Chinese Wall Property (CWP) [2], which forbids an agent interacting with a company to have direct or indirect interactions at a later stage with a competitor.

Our paper is organized as follows. Sections II and III introduce session systems and their semantics. Section IV shows a restriction for which coverability is decidable. Section V assembles these results to provide effective verification tools to check conflicts of interest and CWPs. Section VI compares our approach with related work on services and orchestrations. The proofs and the complete operational semantics of session systems are provided in an extended version available at [3].

II. Session systems

A session system represents the behaviours of (possibly infinite) sets of agents interacting with each other. Our model has two varieties of specifications. Agent behaviours are described by templates that determine how agents initiate and join sessions. Sessions are described by protocols that describe what happens during the course of a single structured interaction. We assume that any pair of actors in the system can communicate directly and reliably with each other, when required. Hence, every actor of the system can share information with a partner as soon as it knows the identity of this partner. However, we make no further assumption about the way communication is implemented.

The key ingredient of our model is the *session*, a structured interaction among a finite set of *agents* to achieve a goal. An *agent* is an entity in the distributed system, e.g., a customer of an online store, a bank providing financial services online, etc. Agents operate at two levels. At an individual level, they can create, join and kill sessions, or query the system for the existence of a particular kind of session. At a collective level, they communicate with each other by playing different *roles* to define the interaction within sessions. Thus, an agent can participate in an unbounded number of sessions in parallel, and will be the creator (owner) of a subset of them. Each agent manages a finite set of data variables that can be modified locally by the agent or during interactions within a session. In addition to data variables, each agent maintains a finite set of *references* to known agents. These references help in controlling interactions within a given scope.

Agents in the system follow predetermined behaviours, defined by templates called *archetypes*. Archetypes are transition systems with guards, whose moves are labelled either by agent operations to manage sessions (join, kill, create, query) or by assignments of variables. Figure 1 shows an example of archetype: an agent that is an instance of this (arche)type can create a session of type S_1, in which it must play the role 'client', then update variable a, join an arbitrary number of sessions of type S_2, and finally kill all sessions of type S_1 that it has created, provided the value of a is tt. We will detail later the meaning of these transitions.

As mentioned earlier, sessions are structured interactions involving a finite set of participating agents. Generic patterns of interactions are called *protocols* and are defined in terms of *roles* (e.g., client, merchant). *Sessions* are instances of these protocols in which roles are instantiated at runtime. We will say that a session s is *of type* P when s is an instance of a protocol P. An agent can thus be involved in several sessions, and play different roles in each of them. For instance, an agent can be a client in one instance of a client-server protocol, and a server in another instance.

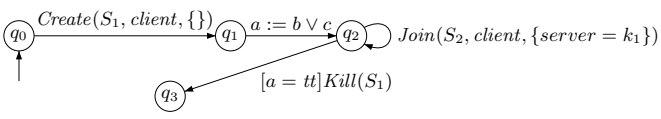
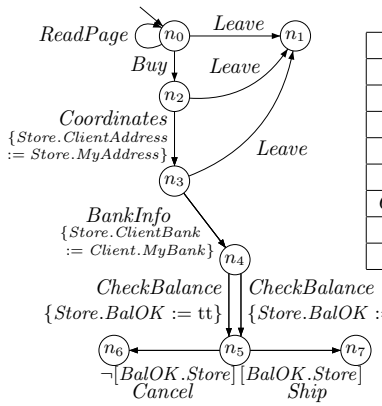


Figure 1. An example of an archetype

A protocol is a finite transition system labeled by guarded shared actions. An action is executed by a subset of roles participating in the protocol. This allows us to model not only synchronous or multicast communications (where an action is located on two or more agents), but also asynchronous message passing (where sending and receiving messages are separate actions, each executed by a single agent). In addition, an action involving a set of roles can assign values to the variables of the agents playing these roles. This allows agents to exchange data values and agent identities.

Figure 2 shows an example of a protocol representing an online sale with three roles, $\{C, S, D\}$, denoting, respectively, a customer, a store, and a delivery service. The shared actions are {`Leave`, `Buy`, `ReadPage`, `Coordinates`, `BankInfo`, `CheckBalance`, `Cancel`, `Ship`}. Variables of the system include {`ClientAddress`, `myAddress`, `ClientBank`, `Mybank`, `BalanceOK`}. The table summarizes how actions are shared. To simplify the example, we have abstracted the answer from the banking system through the action `CheckBalance` that non-deterministically sets the status of a client's account in the store. The meaning of this protocol is rather standard: a customer browses a website, then decides to buy, in which case he has to enter his personal information and bank coordinates. The webstore checks if the payment is granted by the customer's bank and then asks for delivery of the chosen goods. Otherwise, it cancels the transaction. At any stage before payment, the customer can leave the transaction.

A session may start as soon as it is created, even if all its roles are not yet assigned. Consider a protocol modeling a chat service with three roles, one server and two clients. A chat session can be established as soon as one client and the server are ready. The second user may join an existing session later. However, as we allow shared actions among



	C	S	D
ReadPage	+	+	
Leave	+	+	
Buy	+	+	
Coordinates	+	+	
BankInfo	+	+	
CheckBalance		+	
Ship	+	+	+
Cancel	+	+	

Figure 2. An example of a protocol

sets of roles, a shared action can occur only when all roles that participate in it have been assigned to agents.

A session system manages an arbitrary number of parallel sessions with an unbounded number of client requests. Handling multiple sessions at the same time raises security issues, and one has to take side effects into account when performing an action within a session. For instance, the example of Figure 2 may seem correct when considering a single session, or when considering that sessions are handled one after the other. However, if several sessions coexist in the system, the balance *should not* be stored in a global variable by the online merchant. Such problems call for automated verification tools, which is the focus of this paper.

We now formalize the notions of archetypes and protocols. As mentioned earlier, each agent maintains a finite set of data variables, assumed to range over finite domains. For simplicity, we consider only boolean variables, since finite domains can be encoded using combinations of boolean values. We assume that all agents have the same set of (boolean) data variables and denote this set by $\mathcal{V}$. We assume that $\mathcal{V}$ contains a special variable $blocked$, to encode the status of an agent. In addition, each agent has a finite set of *reference variables*, or just *references*, that point to other agents in the system. As with data variables, we assume that all agents have the same set of reference variables, denoted $\mathcal{K}$. *Boolean expressions* over $\mathcal{V} \cup \mathcal{K}$ is defined as follows:

$$\begin{aligned}\phi ::= \;& tt \mid ff \mid v \mid \neg v \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \\ & k_1 = k_2 \mid k_1 \neq k_2 \mid \phi_1 = \phi_2 \mid \phi_1 \neq \phi_2,\end{aligned}$$

where $v \in \mathcal{V}$, and $k_1, k_2 \in \mathcal{K}$, and ϕ_1, ϕ_2 are expressions. Note that negations have been pushed inwards and only conjunctions and disjunctions are allowed at the top level.

Variables and references can be updated through *assignments*. Assignments are denoted by $b := e$, where $b \in \mathcal{V}$ and e is a boolean expression over $\mathcal{V} \cup \mathcal{K}$ (boolean assignment), or by $k := k'$, where $k, k' \in \mathcal{K}$ (reference assignment). In general, given a set of variables $\mathcal{X}$, we write $Expr(\mathcal{X})$ for the set of boolean expressions over $\mathcal{X}$ and $Asg(\mathcal{X})$ for the set of valid assignments to $\mathcal{X}$. As usual, a *valuation* for $\mathcal{X}$ is a function $val()$ that maps each variable in $\mathcal{X}$ to an element of its domain. Given a valuation $val()$ and an expression $e \in Expr(\mathcal{X})$, we write $val() \models e$ if e evaluates to tt under the valuation $val()$. We say that expression e is satisfiable if there exists such a valuation for $\mathcal{X}$.

We are now ready to formally define session systems. We define Σ, the set of *agent operations*, as follows.

$$\begin{aligned}\Sigma = \;& a \mid \quad Spawn(s, r, c) \mid Kill(s) \mid Join(s, r, c) \mid \\ & Bjoin(s, r, c) \mid Query(s, c),\end{aligned}$$

where $a \in Asg(\mathcal{V} \cup \mathcal{K})$, s is the name of a protocol, r is a role in s that the agent intends to play and c is a constraint on the identity of other agents within this protocol. Constraints are boolean combinations of atoms of the form $r_p = r_q.k_j$ in which r_p, r_q are roles of protocol s and $k_j \in \mathcal{K}$. Such a constraint expresses the fact that role r_p must be played by the agent referred to as k_j by the agent playing role r_q in s. This allows an agent to ask to join a session with a specific agent known to it, or, conversely, forbid undesirable agents that it knows about from participating in sessions it creates. We denote by $Cnst(s, \mathcal{K})$ the finite set of possible constraints on roles of a protocol s using variables $\mathcal{K}$.

We now explain the agent operations in Σ. $Spawn(s, r, c)$ creates a session of type s in which the agent creating the session plays role r, and the remaining roles can only be assigned to agents satisfying the constraint $c \in Cnst(s, \mathcal{K})$. $Join(s, r, c)$ asks to join a session of type s, where the requesting agent wishes to play role r. This join request is kept in a set of pending requests and the agent can proceed to perform other operations. $Bjoin(s, r, c)$ is similar to $Join(s, r, c)$, but blocks the requesting agent until an existing session of the desired type is joined. $Query(s, c)$ asks if there is a current session of type s satisfying constraint c and blocks the agent so long as there exists no such session. When an agent is blocked, its variable *blocked* is set to tt. $Kill(s)$ kills all sessions of type s in the system that are owned by the agent executing the action.

Definition 1: An *archetype* is a tuple of the form (Q, q_0, Δ), where Q is a finite set of states, with q_0 the initial state, $\Delta \subseteq Q \times Expr(\mathcal{V} \cup \mathcal{K}) \times \Sigma \times Q$ is a transition relation, with Σ a set of agent operations.

A transition in Δ is a tuple (q, g, σ, q'), where q, q' are states of the archetype, $g \in Expr(\mathcal{V} \cup \mathcal{K})$ is a guard, and $\sigma \in \Sigma$ an agent operation. Archetypes describe how sessions are handled at a high-level by agents. An agent can move from one state to another via a transition and perform the associated agent operation provided the guard of the transition holds in the current valuation. The complete operational semantics of $Spawn$, $Join$, $Bjoin$, $Query$ and $Kill$ is given in the extended version [3].

Next, we define templates for structured interactions between agents, called *protocols*. A protocol S is an automaton whose transitions are labeled by shared actions. The participants in an action are a subset of the roles $\mathcal{R}$ associated with the protocol. We write $r.v$ to refer to the variable v attached to role $r \in \mathcal{R}$. Hence, protocols are transition systems whose guards and assignments are defined over $\mathcal{R}.\mathcal{V} = \{r.v \mid v \in \mathcal{V}, r \in \mathcal{R}\}$ and $\mathcal{R}.\mathcal{K} = \{r.k \mid k \in \mathcal{K}, r \in \mathcal{R}\}$.

Definition 2: A *protocol* is a tuple $S = (N, n_0, \delta, \mathcal{R}, \Gamma, l)$, where

- N is a finite set of session nodes, n_0 is an initial node,
- Γ is a finite alphabet of actions,
- $\mathcal{R} = \{r_1, \ldots r_n\}$ is a finite set of roles,
- $l : \Gamma \times \mathcal{R} \to \{\bot, +, \top\}$ is a function that indicates whether role r participates in the shared action γ and whether this action is the last one performed by this agent in the current session. More precisely, $l(\gamma, r) = \bot$ if role r does not participate in γ, $l(\gamma, r) = \top$ if role r participates in γ and γ is its last action in the current

session, and $l(\gamma, r) = +$ if role r participates in γ but γ is not its last action in the current session.
- $\delta \subseteq N \times Expr(\mathcal{R}.\mathcal{V} \cup \mathcal{R}.\mathcal{K}) \times \Gamma \times 2^{Asg(\mathcal{R}.\mathcal{V} \cup \mathcal{R}.\mathcal{K})} \times N$ is a transition relation.

Sessions are instances of protocols with concrete agents assigned to roles. As all roles need not be defined simultaneously, a session may start with some roles still unassigned. A transition (n, g, γ, as, n') in the protocol indicates that the session can move from node n to n' through the shared action γ provided guard g holds with respect to the current values of $\mathcal{R}.\mathcal{V} \cup \mathcal{R}.\mathcal{K}$ and all roles involved in γ (i.e., r such that $l(\gamma, r) \in \{+, \top\}$) have been assigned to agents. Executing γ results in the update of variables: the variables $R.\mathcal{V} \cup R.\mathcal{K}$ are modified as described in the set of assignments as. Note that a shared action allows multiple assignments, performed atomically, in parallel. Practically, performing actions involving more than one agent would require use of a shared memory, or synchronization among participants of the session. For ease of implementation, one could also require actions to be local to a single agent, and communications to be asynchronous. However, these implementation details do not affect the decidability results described in this paper, and are left for future work.

Definition 3: A *session system* is a pair $SS = (\mathcal{A}, \mathcal{S})$ where $\mathcal{A} = \{A_1, \ldots, A_k\}$ is a finite set of archetypes and $\mathcal{S} = \{S_1, \ldots, S_q\}$ is a finite set of protocols. Each archetype in $\mathcal{A}$ is of the form $A_i = (Q_i, q_0^i, \Delta_i)$, and each protocol in $\mathcal{S}$ is of the form $S_j = (N_j, n_0^j, \delta_j, \mathcal{R}_j, \Gamma_j, l_j)$.

III. Session systems semantics

In this section, we describe the operational semantics of session systems. A *configuration* of a session system describes the local states of all agents and sessions, together with the valuation of their variables. We represent a configuration as a (vertex and edge) labeled graph $C = (V, E, \tau)$ where

- $V = V_A \uplus V_S$ is a finite set of vertices, where V_A denotes agents (i.e., instantiations of archetypes) and V_S denotes sessions (i.e., instantiations of protocols).
- $E \subseteq (V_A \times \mathcal{K} \times V_A) \cup (V_A \times \mathbb{N} \times \{tt, ff\} \times V_S) \cup (V_A \times \{qry\} \times V_S)$ is a set of labeled edges over V, representing agent references, and connections between agents and sessions. A triple $(v, k, v') \in V_A \times \mathcal{K} \times V_A$ means that the agent represented by v has stored the identity of the agent represented by v' via reference variable k. An edge of the form (v, qry, v') means that the agent represented by v is querying the system for a session whose characteristics are described by v'. The number of labels for these two kinds of edges is finite. Edges of the form (v, l, s) where $l = (n, b) \in \mathbb{N} \times \{tt, ff\}$, $v \in V_A$, and $s \in V_S$ indicate one of the following.
  - The agent represented by vertex v plays role n in a session represented by vertex s. The boolean flag b is set to tt if v is the owner of the session, and to ff otherwise.
  - The agent represented by v asks to play role n in a session. Vertex s then represents a join request. The label of this vertex defines the type of session that agent v wishes to join, along with the constraints attached to this request. The flag b is set to ff (v cannot own a joined session that it did not create).

  Note that an agent can be connected via reference edges only to a finite number of agents, but it can be connected to an unbounded number of session vertices. Conversely, a session can only be connected to a finite number of agents that is bounded by the number of roles in the protocol it instantiates.
- τ labels each vertex of V_A with details of the agent it represents (archetype, control state, valuation of variables) and each vertex of V_S with details of the session it represents (protocol name, current node, constraints on roles yet to be instantiated).
  More formally, the labelling of vertices is defined as follows. We have $\tau : V_A \to \mathcal{A} \times \bigcup_{i \in \mathcal{A}} Q_i \times Val(\mathcal{V})$. The label $\tau(v) = (A_i, q, val)$ indicates that v represents an agent behaving according to archetype A_i, currently in state $q \in Q_i$, with valuation val of its variables. Let us now consider the labeling of session vertices. We have $\tau : V_S \to \mathcal{S} \times \mathcal{N} \times Cnst(\mathcal{S}, \mathcal{K}) \times 2^{\mathcal{R}}$, where $\mathcal{N}$ is the union of all possible nodes appearing in protocols. For every vertex $v \in V_S$ that represents an instance of a running session, we have $\tau(v) = (s, n, c, J)$, which indicates that v is an instance of protocol s, currently in node n with contraints c attached to uninstantiated roles, such that roles in J have not yet been assigned. Similarly, $\tau(v) = (s, c)$ if v represents a join request or a query for a protocol s with constraints defined by c. In these cases, vertex v is connected to an agent vertex either via an edge labeled by (r, ff), for a join request, or via an edge labeled by qry, for a query.

Clearly, the set of labels attached to edges and vertices is finite. We further impose well-formedness constraints. For every vertex $v \in V_S$, if $\tau(v) = (s, c)$ then there exists a single vertex $v \in V_A$ connected to v. If $\tau(v) = (s, n, c)$, then the number of agents connected to v is at most the number of roles in protocol s. Further, exactly one agent is connected to a session via an edge labeled by (n, tt), i.e., each session has only one owner.

Figure 3 shows a graphical representation of a possible configuration for a set of archetypes $\mathcal{A} = \{A_1, A_2\}$ and a set of protocols $\mathcal{S} = \{S_1, S_2\}$. Agents are represented as rectangles and sessions as ellipses. For clarity, we have not represented variable valuations in the labels of agents. In this configuration, agent p of (arche)type A_1 is in state q_1, agent q of type A_1 is in state q_3, and agent r of type A_2 is in state q_2. Agent p knows agent q through its reference variable k_1. Agents p, q, r are involved in a session of type S_1 owned by q, currently in state n_2, where they play roles

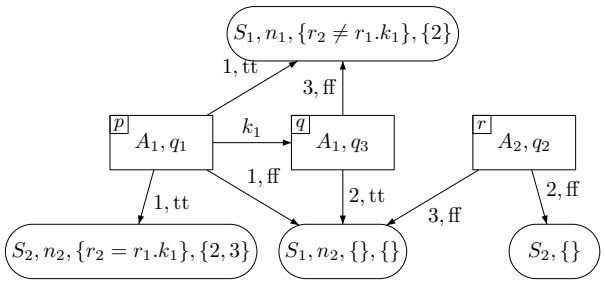


Figure 3. Graphical representation of configurations

r_1, r_2, r_3, respectively. Agent p is the owner and plays role r_1 in an instance of session protocol S_2, currently in node n_2. A constraint indicates that role r_2 can only be assigned to the agent known as k_1 by p, and roles r_2, r_3 are not yet assigned. Agents p, q have joined a session of type S_1 and play roles r_1, r_3, respectively, within the session. Agent p is the owner of this session, and the constraint says that the agent that will join role r_2 should differ from the agent known as k_1 by p. Agent r has asked to join an existing session of type S_2 in role r_2.

In the rest of the paper, we denote by $\mathcal{C}$ the (possibly infinite) set of all configurations. We provide an informal semantics for our session systems. A session system starts in an *initial configuration* C_0 (typically the empty configuration). Three kinds of moves can occur from a configuration: a local move of an agent, a shared action within a session, or an environment move, i.e., either an operation that matches a join request to an existing session or existing query, or the arrival of a new agent in the system. Local agent moves can change the values of the variables owned by the agent, create a new session, ask to join a particular kind of session, kill sessions owned by the agent, or modify its references. All these actions are described in our model through a finite number of creations or deletions of vertices and edges, and a relabeling of vertices, due to a change in the valuation and control state of the agents. *Spawn* creates a new session vertex and connects it to the creating agent. Each form of join (*Join*, *Bjoin*) and query also creates a new vertex and a new edge with appropriate labels corresponding to the operation. *Kill* suppresses edges and session vertices corresponding to some sessions owned by the agent. As all these operations are transitions of an archetype, this also results in a relabeling of the agent vertex to model the change of state. Session actions model interactions among agents within a session: in some currently active session, a shared action that is currently enabled is performed. Such a move results in a relabeling of a session vertex in a configuration (the session changes its state), a relabeling of agents involved in the session (the agents' variables may be updated), and a finite number of changes in the edges representing the references of agents contributing to the session.

Environment moves are high-level actions provided by a system that manages all sessions. We do not detail how such a system is implemented. Arrival of a new agent introduces a fresh instance of some archetype into the system (it creates a new agent vertex). Agent vertices are never removed. Query unlocking simply consists of removing the edge and vertex modeling a query, and updating the status of the special variable $blocked$ in the querying agent. The last kind of move is servicing a join request. This operation *nondeterministically* matches a pending join request and an available session. It results in the suppression of the vertex representing the request, a connection of the joining agent to a session vertex with an appropriate label, and in a relabeling of the vertex representing the joined session to take into account the constraints imposed by the joining agent. Note that join requests can remain pending for an arbitrarily long time, even if matching sessions are available. There is no obligation to match pending join requests "eagerly".

Following this informal definition of the semantics of session systems, we can define the behaviour of a session system as a sequence of moves satisfying the semantics. A move from a configuration C to a configuration C' via action σ is denoted by $C \xrightarrow{\sigma} C'$. A *run* of a session system from a configuration C_0 is a sequence of moves $\rho = C_0 \xrightarrow{\sigma_1} C_1 \ldots \xrightarrow{\sigma_k} C_k$. Given a set of configurations X, we will say that ρ is a *run over* X if it is exclusively composed of configurations from X. With this semantics, a session system defines an infinite state transition system. The formal semantics of moves is defined in detail in the extended version [3]. We say that a configuration C is *reachable* from an initial configuration C_0, denoted $C_0 \longrightarrow C$, if there exists $\rho = C_0 \xrightarrow{\sigma_1} C_1 \ldots \xrightarrow{\sigma_n} C$.

Theorem 4: Reachability of a configuration C from an initial configuration C_0 is undecidable.

A similar result was proved in [1] for a less expressive session model. Session systems can simulate reset Petri nets, for which reachability is undecidable: sessions are used to encode place contents, adding tokens is simulated by creating sessions, consuming them is simulated by entering a session and terminating it, and resetting places is simulated by killing sessions corresponding to the reset places.

IV. Well-structured Session Systems

As observed above, session systems have an infinite space of configurations for which even reachability is undecidable. However, we now show that under some mild restrictions, these systems are well-structured, which implies that some interesting problems such as coverability become decidable. In the next section, we will show that this allows us to verify business rules such as conflict of interest.

We start with some relevant notions and results from [4]. A *well-quasi ordering* (WQO) on a set X is a reflexive, transitive binary relation $\leq$ such that any infinite sequence $x_0, x_1, \ldots$ of elements of X contains a pair x_i, x_j such that $i < j$ and $x_i \leq x_j$. In fact, $\leq$ is a WQO iff it is well-founded on X (i.e., it does not contain infinite decreasing sequences)

and does not contain infinite antichains, i.e., infinite sets of incomparable elements. The *upward closure* of a set X is $\uparrow X = \{y \mid \exists x \in X, y \geq x\}$. A set X is called *upward closed* if $\uparrow X = X$. Any upward closed set X in a WQO can be represented by a *finite basis* $B(X) = \min_{\leq}\{X\}$.

A *well-structured transition system (WSTS)* is a structure $(X, succ, \leq)$ where X is a possibly infinite set of elements, $succ \subseteq X \times X$ is a transition relation, and $\leq$ is a preorder on X such that $(X, \leq)$ is a wqo, and $succ$ satisfies the following monotonicity property: $\forall x \in X, (x, x') \in succ$ and $y \geq x$ implies $(y, y') \in succ^*$ for some $y' \geq x'$, where $succ^*$ is the transitive and reflexive closure of relation $succ$. For an upward closed set X, $pre(X) = \{y \mid \exists x \in X, x \in succ(y)\}$ denotes the set of predecessors of X, with $pre^*(X) = \{y \mid \exists x \in X, x \in succ^*(y)\}$. The *pred-basis* of X is the finite set $pred_B(X) = B(pre(X))$. We say that a WSTS has an *effective pred-basis* if there exists an algorithm that accepts an element x and returns a basis for $\uparrow Pre(\uparrow x)$. We recall the following result (see for instance [4]):

Proposition 5 ([4]): Let $S = (X, succ, \leq)$ be a WSTS with decidable $\leq$ and effective pred-basis. Then, for a pair $x, x_0 \in X$, one can decide *the coverability problem*, which asks whether there exists a run of S from x_0 to some x' such that $x' \geq x$.

To apply Proposition 5 to sessions systems, we define an ordering on configurations.

Definition 6: Let $C_1 = (V_1, E_1, \tau_1), C_2 = (V_2, E_2, \tau_2)$ be two configurations. We say that C_1 is a *subgraph* of C_2, denoted by $C_1 \sqsubseteq C_2$ iff there exists a pair of injective mappings $\psi : V_1 \to V_2$ and $\psi' : E_1 \to E_2$ such that:

- $\forall v \in V_1, \tau(\psi(v)) = \tau(v)$
- $\forall e = (v, l, v') \in E_1$, with $l \in \mathcal{K} \cup (\mathbb{N} \times \{tt, ff\}) \cup \{qry\}$, $\psi'(e) = (\psi(v), l, \psi(v))$ is an edge of E_2.

We say that two configurations $C_1, C_2 \in \mathcal{C}$ are *isomorphic* iff $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_1$. As configurations are finite graphs, *one can effectively check if* $C_1 \sqsubseteq C_2$. Now if $(\mathcal{C}, \longrightarrow, \sqsubseteq)$ were a WSTS, we could directly check properties of session systems using Proposition 5. However, $(\mathcal{C}, \sqsubseteq)$ *is not* a WQO: one can design sets of pairwise incomparable configurations of unbounded size. To overcome this problem, we need to restrict the set of configurations considered.

One obvious restriction is to bound the number of agents in configurations. In [1], it is shown that this suffices to obtain a WQO on configurations (roughly speaking, the set of configurations can be encoded as an integer vector counting the number of occurrences of each tuple ⟨session, state, unassigned roles⟩). However, for some applications, this can be rather restrictive. For instance, a webstore is supposed to accept huge numbers of clients. Even if its resources call for a bound on the number of clients using a site, this bound depends on architectural choices, and not on the behavioural specification of the system. Increasing the resources may increase the number of clients a store can handle, and one cannot set such a bound a priori.

In this paper, we propose a different restriction, namely *k-boundedness* of configurations, which allows us to model systems with unboundedly many agents and sessions, and yet obtain a WQO on configurations.

Let $C = (V, E, \tau)$ be a configuration. We say that a vertex $v_2 \in V_A$ is a successor of vertex $v_1 \in V_A$ if there is a reference edge from v_1 to v_2 (i.e, $(v_1, k_i, v_2) \in E$ for some $k_i \in \mathcal{K}$) or there exists a pair of edges $(v_1, l, s), (v_2, l', s) \in E$ connecting the two agent vertices to the same session vertex s. A *path* in C is a sequence of vertices $v_1, v_2 \ldots v_n$ such that v_{i+1} is a successor of v_i in C. This path is simple if $i \neq j$ implies $v_i \neq v_j$. The length of a simple path is the number of vertices it contains. A configuration C is *k-bounded* if it has no simple paths of length greater than k.

We will denote by $\mathcal{C}^k$ the set of k-bounded configurations. Note that $\mathcal{C}^k$ is not a finite set: configurations of $\mathcal{C}^k$ may contain an arbitrary number of k-bounded connected components, containing arbitrary numbers of sessions. So, the number of vertices in configurations of $\mathcal{C}^k$ is not bounded, but the way these vertices are connected is constrained.

The two ways a configuration $C \in \mathcal{C}^k$ can evolve into $C' \notin \mathcal{C}^k$ are through a *merge* move by the environment, or by an agent learning the identity of a new agent, which creates a new simple path of length greater than k. In such a situation, that is, when $C \xrightarrow{\sigma} C'$ with $C \in \mathcal{C}^k$ and $C' \in \mathcal{C}^{k' > k}$ for some action σ, we consider that the system leaves the set of accepted configurations, and we replace this move by a move $C \to \top$ to a special configuration $\top$ ("Top") that has the following properties:

- $C \sqsubseteq \top$ for all configurations $C \in \mathcal{C}^k \cup \{\top\}$. Henceforth we write $\mathcal{C}^k_\top$ for $\mathcal{C}^k \cup \{\top\}$.
- All types of moves (session, agent, environment) are enabled at $\top$ and the resulting configuration is $\top$.

The element $\top$ should not be considered as a real configuration, but rather as an indication that a run of the system has reached a configuration that is not k-bounded. With this restriction we obtain the well-structured property.

Theorem 7: For any fixed $k \in \mathbb{N}$, $(\mathcal{C}^k_\top, \longrightarrow, \sqsubseteq)$ is a well-structured transition system.

Now, in order to use Proposition 5 to show decidability of coverability for session systems, it remains to prove that the computation of a pred-basis is effective. In the rest of the paper, we will use set saturation techniques, and represent an upward closed set using its basis. Indeed, as $\mathcal{C}^k_\top$ is a WQO, any upward closed set of configurations has a *finite* basis: for a configuration C in $\mathcal{C}^k_\top$, $\{C\}$ is a finite basis for $\uparrow C$. This property extends to arbitrary sets of configurations. For any finite $\{C_1, C_2, \ldots, C_n\} \subseteq \mathcal{C}^k_\top$, the basis of $\bigcup_{i=1}^n \uparrow C_i$ is the set of minimal elements w.r.t. $\sqsubseteq$ in $\{C_1, C_2, \ldots, C_n\}$. Using bases as representations for upward closed sets in a WQO setting allows us to work with finite representations of infinite sets. Further, it is sufficient to manipulate a basis $B = \{B_1, B_2, \ldots, B_n\}$ to decide membership. If X is an upward closed subset of $\mathcal{C}^k_\top$ represented by its finite basis

$\{B_1, B_2, \ldots, B_n\}$, then checking that $C \in X$ amounts to checking $B_i \sqsubseteq C$ for some B_i in the basis of X.

For a configuration $C \in \mathcal{C}^k_\uparrow$, we define $Pre(C) = \{C' \mid C' \xrightarrow{\sigma} C\}$, as usual. This extends to the upward closure as follows: $Pre(\uparrow C) = \{C' \mid C' \xrightarrow{\sigma} C'' \sqsupseteq C\}$. For a set of configuration $X \subseteq \mathcal{C}^k_\uparrow$, we have, as usual, $Pre(X) = \bigcup_{C \in X} Pre(C)$ and $Pre(\uparrow X) = \bigcup_{C \in X} Pre(\uparrow C)$. It is easy to see that a basis and pred-basis for an upward closed set are effectively computable. Thus, we have:

Lemma 8: Let X be an upward closed subset of $\mathcal{C}^k_\uparrow$ represented by its basis $\{B_1, B_2, \ldots, B_n\}$. Then we can effectively compute a basis for $\uparrow Pre(X)$.

In the context of session systems, the *coverability* problem consists of deciding if, from an initial configuration C_0, one can reach a configuration C' that covers a target configuration C (i.e., such that $C \sqsubseteq C'$). When C is coverable from C_0, we write $C_0 \rightsquigarrow C'$. Similarly, we will say that a run $\rho = C_0 \xrightarrow{\sigma_1} C_1 \ldots \xrightarrow{\sigma_k} C_k$ covers C if $C \sqsubseteq C_k$. From the properties of WSTS, Theorem 7 and Lemma 8 we have:

Corollary 9: For a session system $S = (\mathcal{A}, \mathcal{S})$, coverability of a configuration C by a run over $\mathcal{C}^k_\uparrow$ from an initial configuration C_0 is decidable.

The complete proof is available in [3], but we describe here an effective algorithm to check, for a given configuration C and session system S, if there exists a run of S starting from configuration C_0 that reaches a configuration that subsumes C. The algorithm is a set-saturation technique that computes a basis for $\uparrow pre^*(\uparrow C)$ as a fixpoint. We start from $PB_0 = B(\uparrow C) = \{C\}$, and compute iteratively $PB^k = PB^{k-1} \cup B(\uparrow pre(\uparrow PB^{k-1}))$, until $\uparrow PB^k = \uparrow PB^{k-1}$. It has been proved in [4] that for any upward closed set, this algorithm is correct and terminates when the pred-basis computation is effective. Hence, at the end of the fixpoint computation, we have $\uparrow \bigcup PB^k = Pre^*(\uparrow C)$. Once basis PB^k is built, we compare its elements to C_0 to check if C is coverable from C_0.

V. Model checking

We now use the results of Section IV to check business rules on session systems. As mentioned in the Introduction, session systems can model business processes, web-based applications, and transactional systems. For these applications, several properties are of practical interest. In this paper, we focus on two classes of interesting properties, namely, conflicts of interest and Chinese Wall Properties.

A. Conflicts of interest

Conflicts of interest may arise when two clients of a webstore are involved at the same time in an online purchase involving the same item. Similarly, one may not want a bank to deliver its services to competing business entities. We formalize such situations as undesirable patterns, i.e., partial descriptions of undesirable configurations and show that we can check whether such undesired configurations can occur during the lifetime of a system.

Definition 10: A *pattern* is a graph (V, E, τ), where V, E and τ have the same interpretation as in configurations.

We say that a configuration C *matches* a pattern P if $P \sqsubseteq C$. Further, a run $\rho = C_0 \xrightarrow{\sigma_1} C_1 \ldots \xrightarrow{\sigma_k} C_k$ of a session system *meets* a pattern P if some configuration C_i along the run matches P. Finally, a pattern is *reachable* in a session system SS from a configuration C_0 if there exists a run of SS that meets P. Thus, checking for a *conflict of interest* (modeled as a pattern) corresponds to checking if it is possible to reach a configuration which matches the pattern. Note that patterns need not be configurations. However, decision procedures for coverability can be used for patterns, as saturation techniques and proofs for well structure and effectiveness do not use the fact that the considered objects are configurations: $\sqsubseteq$ is defined for any kind of subgraph, and we can set $P \sqsubseteq \top$ for every pattern P. Similarly, upward closure of patterns, predecessors, bases, can be defined for patterns, and proved effective. As we restrict ourselves to k-bounded configurations, we also restrict to k-bounded patterns for a fixed k and denote by $\mathcal{P}^k$ the set of k-bounded patterns. Now, the results on coverability extend to patterns:

Proposition 11: Given a session system SS, pattern $P \in \mathcal{P}^k$, and configuration $C_0 \in \mathcal{C}^k$, one can decide if there exists a run of SS over $\mathcal{C}^k$ starting from C_0 that meets P.

Proof Sketch: Following the proof of Corollary 9, we can build a finite sequence $PB^0, \ldots, PB^n$ of unions of bases such that $\uparrow PB^n = \uparrow pre^*(\uparrow P)$. Using the steps of the construction of PB^n, we build a directed acyclic graph G_{PB}, whose vertices are the bases computed at each step. An edge connect bases b and b' if b is computed at step j and b' at step $j-1$. It now suffices to examine paths of G_{PB} to decide existence of a run over $\mathcal{C}^k$ that meets P. ■

Figure 4 illustrates the construction of the pred-basis, and of the graph G_{PB}. Circles labeled B represent a temporary basis computed during one step of the algorithm. Edges represent the fact that a temporary basis was computed as a predecessor of a formerly discovered one.

B. Chinese Wall Properties

Conflicts of interest represent properties of systems at a single instant. Business rules need to guarantee properties

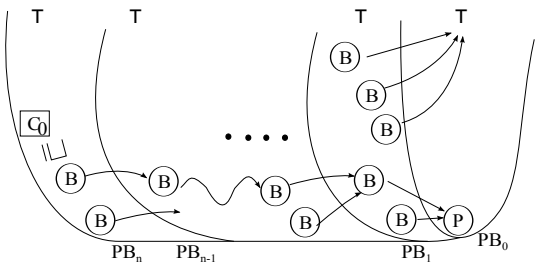


Figure 4. Computing a Basis for $Pre^*(\uparrow C)$ or $Pre^*(\uparrow P)$

throughout the lifetime of a system by considering several situations that should (or should not) appear along a run. For instance, we may want to ensure that a client receives an item sold by a webstore only after making the payment. One may also want to enforce non-competition clauses, i.e., an agent involved in an activity should not provide the same service at a *later* date to a competitor (such properties are also called Chinese Wall Properties, or CWP [2]). In both examples, a first attempt would be to model the properties as a pair of coverability problems: can one reach a pattern P_2 from a configuration matching P_1 that is accessible from C_0? This approach is flawed: in this approach, P_1 and P_2 may be witnessed by different agents and sessions, while the sales or non-competition examples depict situations involving the same participants. To model such situations, we need to specify elements that refer to the same agents in both patterns P_1 and P_2. This is formalized as follows:

Definition 12: A *correlated pattern* is a triple (P_1, P_2, ψ) where, for $i = 1, 2$, $P_i = (V_i, E_i, \tau_i)$ is a pattern and $\psi : V_1 \rightharpoonup V_2$ is a partial function.

A run $\rho = C_0 \to C_1 \dots C_n$ *matches* a correlated pattern (P_1, P_2, ψ) if there exist C_i and C_j with $0 \leq i < j \leq n$ and embeddings h_1, h_2 such that $P_1 \sqsubseteq C_i$ via embedding h_1, $P_2 \sqsubseteq C_j$ via embedding h_2 and for all $v \in dom(\psi)$, $h_1(v) = h_2(\psi(v))$. In other words, vertices in the two patterns that are connected by ψ must map to the *same* vertex in the corresponding configurations where the two patterns are embedded. A *Chinese Wall Property (CWP)* is specified by a finite set of correlated patterns $\{(P_i, P'_i, \psi_i)\}_{i \in 1 \dots n}$. A session system *violates* a CWP $\{(P_i, P'_i, \psi_i)\}_{i \in 1 \dots n}$ if one of its runs matches one of the correlated patterns (P_j, P'_j, ψ_j). CWPs allow us to specify sets of incompatible situations that should not occur: an employee holding a counselor position at time t in a bank A does not have the right to hold the same position at any time greater than t, nor to be head of client accounts department for a competitor bank B.

In the rest of this section, we will prove that checking the violation of CWPs is decidable for k-bounded session systems. More formally:

Theorem 13: Let SS be a session system, $k \in \mathbb{N}$ be a bound on path length in configurations of SS. Let $C = \{(P_i, P'_i, \psi_i)\}_{i \in 1 \dots n}$ be a CWP. Then, one can decide if there exists a run of SS over $\mathcal{C}^k$ violating C.

We prove this theorem as follows. First, as we need to identify agents in patterns, we will slightly adapt the semantics of session systems. When an agent is created, its vertex will be attached a tag that will never be modified, and will help to trace it along a run. This results in a slight change in the semantics preserving WQO and compatibility. Then we prove that CWP verification amounts to checking two coverability problems with tagged patterns.

Let $T_\perp = T \cup \perp$, where T is a finite set of *tags*. Define an ordering $<$ on $T_\perp$ as follows: $\perp < t$ for all $t \in T$. In other words, elements of T are not ordered with respect to each other. A T-tagged configuration is a graph $C = (V, E, \tau, Tg)$, where V and E are vertices and edges as in configurations and $Tg \subseteq 2^T$ is the set of tags used in C. For edges and session vertices, the labeling τ is defined as for configurations. For $v \in V_A, \tau(v) \in L \times T_\perp$, where L is the set of labels defined for configurations. We extend the ordering $<$ on $T_\perp$ to $L \times T_\perp$ such that $(\ell, \perp) < (\ell, t)$ for all $\ell \in L$, and (ℓ', t) and (ℓ, t) are incomparable. For a pair of tagged configurations C_1, C_2, we write $C_1 \sqsubseteq_t C_2$ if there exists a homomorphism h such that for all $v \in V_1$, $\tau_1(v) < h(\tau_2(v))$, and $Tg_1 = Tg_2$. That is, configurations that use different sets of tags from T are incomparable, and configurations with incomparable sets of used tags are also incomparable.

For a tagged configuration $C_t = (V, E, \tau)$, we define its untagged version $UT(C_t) = (V, E, \tau')$ where for every vertex v in $V_A, \tau'(v) = \ell$ iff $\tau(v) = (\ell, x)$, and every vertex v in $V_S, \tau'(v) = \tau(v)$. Let $T\mathcal{C}^k$ be the set of all T-tagged configurations whose untagging is in $\mathcal{C}^k$ and let $T\mathcal{C}^k_\top = T\mathcal{C}^k \cup \top$. We slightly update the semantics of moves from $\to$ to $\to_t$ to take tags into account. The only change is the environment move *create*. When adding a new agent, the create action now assigns a random unused element from $T_\perp$ as a permanent tag for the new agent and adds it to the set of used tags. The new agent will carry this tag for the rest of the run: for every $C_1 \to C_2$, $v \in V_1 \cap V_2$ and $\tau_1(v) = (\ell, t)$ implies that $\tau_2(v) = (\ell', t)$ for some ℓ'. Note that tags play no role in enabling or disabling actions. Untagging easily extends to runs, i.e., for any tagged run $\rho = C_0 \to_t C_1 \dots C_k$, $UT(\rho) = UT(C_0) \to UT(C_1) \dots UT(C_k)$. We do not change the semantics of patterns, and say that a configuration C embeds a pattern P if $P \sqsubseteq UT(C)$. We then obtain:

Lemma 14: $(T\mathcal{C}^k_\top, \to_t, \sqsubseteq)$ is a WSTS. Also, if $C \in T\mathcal{C}^k_\top$, then $\uparrow Pre(\uparrow C)$ has an effectively computable finite basis.

We can now explain the connection between tags and correlated patterns. Tags will help identify common agents in correlated patterns. We define *tagged patterns*, i.e., pairs of patterns that carry tags that identify commonalities. Let (P, P', ψ) be a correlated pattern, with $P = (V, E, \tau)$, $P' = (V', E', \tau')$, and let $T_\psi = \{t_v \mid v \in dom(\psi)\}$. The corresponding T_ψ-tagged correlated pattern is (P_t, P'_t, ψ) where:

- $P_t = (V, E, \tau_t)$, where $\tau_t(v) = (\tau(v), t_v)$ if $v \in dom(\psi)$ and $\tau_t(v) = (\tau(v), \perp)$ otherwise.
- $P'_t = (V', E', \tau'_t)$ where $\tau'_t(v) = (\tau'(v), t_v)$ if $v \in range(\psi)$ and $\tau'_t(v) = (\tau'(v), \perp)$ otherwise.
- $\psi(v) = v'$, $\tau_t(v) = (\ell, x)$ and $\tau'_t(v') = (\ell', y)$ imply $x = y$.

Using tagged patterns, we can ensure that an agent v identified by a tag t in pattern P of a configuration during an execution is the same agent (with tag t) in the following configuration meeting another pattern P'. We are now ready to prove decidability of correlated patterns checking:

Lemma 15: There exists a run matching a correlated pattern (P, P', ψ) from C_0 iff the following hold:

(i) there exists $C_t \in Pre^*(\uparrow P'_t)$ with $P_t \sqsubseteq_t C_t$ in the tagged semantics (using tags from T_ψ).
(ii) $C_0 \in Pre^*(UT(\uparrow P_t \cap Pre^*(\uparrow P'_t)))$ with respect to untagged semantics.

This lemma shows that correlated pattern matching reduces to a pair of coverability problems, and suffices to prove Theorem 13. By Lemma 14, coverability of tagged patterns is effective, so part (i) of Lemma 15 is effective. Similarly, $\uparrow P_t \cap Pre^*(\uparrow P'_t)$ can be represented by a finite basis, so part (ii) of Lemma 15 is also effective.

VI. Related Work

Several formalisms have been proposed to implement or *orchestrate* sessions into larger applications. A BPEL [5] specification describes a set of independent communicating agents with a rich control structure. Coordination is achieved through message-passing. Interactions are grouped into sessions implicitly through *correlations*, which specify data values that uniquely identify a session—for instance, an order ID. ORC [6] is a programming language for the orchestration of services. It allows algorithmic manipulation of data, with an orchestration overlay to start new services and synchronize their results. ORC has better mechanisms to define workflows than BPEL, but lacks the notion of correlation required to establish sessions among participants in a service. AXML [7] 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. A common feature of these formalisms is that they aim to describe *implementations* of web services or orchestrations. All of them are Turing powerful, hence properties such as termination of a service, or coverability are undecidable.

To obtain decidability of coverability for session systems, we have used well quasi ordering on graphs, with a restriction to yield well-structuredness. Several papers have explored verification techniques on models based on graph transformation systems (GTS). [8] studies GTS with the graph minor relation and shows that several subclasses are WSTS. However, we cannot use the minor relation when modeling business rules. In our model, patterns describe direct connections between agents and sessions. Graph minors allow collapsing of nodes and edges, describing connectivity of vertices via paths rather than direct edges. Sangnier et al [9] consider reachability and coverability for GTS ordered by subgraph inclusion. Their decidable classes are GTS without deletion rules, context free graph grammars, or grammars with mandatory hyperedge contraction rules. In our model, sessions can end or be killed, hence deletion can occur, and the join operation merges two edges, hence our model is not context free.

A lot of effort has been devoted to modeling services in the π-calculus. *Session types* [10] are a formal model for web services, and have been enhanced to capture various features such as multiple instantiations of identical agents [11] and nested sessions [12]. The main focus is to determine whether an otherwise unconstrained set of processes adheres to the communication discipline specified by a session type. Verification on session types address features such as information flow between agents. The expressive power of the whole π-calculus and session types do not allow for verification of reachability or coverability properties. [13] uses WSTS to show that a fragment of spatial logic is decidable for well-typed π-calculus processes, which can express safety properties. A solution to coverability for depth-bounded π-calculus processes has been proposed in [14]. For this class, a forward coverability algorithm (EEC) terminates, even if the depth bound is not known in advance. Both depth-bounded π-calculus processes and k-bounded configurations can be represented as graphs with bounded path length. Note, however, that k-boundedness for session systems is obtained through a restriction in the semantics. We do not yet have any syntactic restrictions on session systems that ensure a bound on path lengths in configurations.

In the π-calculus, one can view a session as a shared local name. However, some features of session systems do not have straightforward translations into the π-calculus. First, agents can kill sessions. This feature is essential in a web-based system, where a server may shut down and cancel a series of ongoing transactions. In the π-calculus, local names can survive the end of a session, and processes are not meant to be aborted. Second, joining a session and querying the system to find an occurrence of an existing session are non-deterministic actions provided by the environment. Though one could perhaps model an environment and simulate killing, querying, and joining in the π-calculus, the semantics of both models appear quite different.

Several variants of π-calculus have been designed to model services. A variant of ORC and π-calculus is proposed by [15] in which processes communicate via streams. This model is more expressive than session systems, since it allows selecting values from (ordered) streams, but this comes at the cost of decidability. Implicitly, processes can interact but they run until completion, unlike sessions, which can be interrupted. The multiparty session formalism called μ**se** [16] avoids managing session identities explicitly, like session systems. In μ**se**, processes within a session can communicate privately. An arbitrary number of sessions can be created at a site and additional communications are allowed among processes located on the same site. A merge mechanism allows a process to enter a session at any point, and persistent services can be implemented. Unlike session systems, in which each session has a fixed number of participants, μ**se** sessions can allow an arbitrary number of processes to join. However, there is no mech-

anism to abort processes. *CASPIS* [17] is a formalism for orchestration influenced by the π-calculus and by ORC. It provides pairwise sessions, modeled as service calls that create private names shared by the caller and callee of a session, and pipelining—a way for service P to call another service Q whenever a new value in produced by P. Unlike the preceding variants, CASPIS allows guarded sums (i.e., internal choices), and provides a mechanism to terminate sessions. *Conversation types* [18] extend π-calculus and replace channel based communications by context sensitive message based communication. A conversation is a behavioral type describing multi party interactions among processes. [18] provides typing mechanisms to ensure that processes implement conversations in a compatible way and never get stuck when interleaving sessions. A conversation is close to our notion of session but allows for an unbounded number of participants. Like other π-calculus variants, this formalism does not allow conversations to be aborted. *COWS* [19] is a language for service orchestration. COWS allows stateful services with special variables that implement correlations, as in BPEL, a wait operation to suspend processes and a kill operation that terminates terms within a delimited scope. The semantics of kill does not take into consideration the nature of the canceled terms, while in session systems, only the owner can kill a service.

VII. Conclusion

We have presented a formalism to model session-based systems with arbitrary numbers of sessions and agents. A restriction of the semantics to runs over bounded path length configurations allows us to decide coverability problems. An immediate consequence is that we can also check properties such as conflict of interest or Chinese Wall Properties.

Several issues remain. The first is efficiency. Our results rely on well-structured systems and set saturation techniques. We do not know the exact complexity of checking coverability for session systems, but the complexity of WSTS can be very high, and may need to consider runs of non-elementary length. Even with a fixed set of agents, session systems have the expressive power of reset Petri nets [1], for which coverability is Ackermann-hard [20]. Abstraction techniques, such as those proposed in [14] for depth-bounded processes, may help in reducing complexity, and also possibly avoid the k-boundedness restriction. Finally, our model considers finite data. A possible improvement is to introduce well-structured data, either for variables, or as elements conveyed within sessions.

Acknowledgments

This work is an outcome of the DISTOL associated team. We would like to thank the anonymous referees for their useful comments. Last, we dedicate this paper to Philippe Darondeau, who played a key role in the initial formulation of this work.

References

[1] P. Darondeau, L. Hélouët, and M. Mukund, "Assembling sessions," in *ATVA*, ser. LNCS, vol. 6996, 2011, pp. 259–274.

[2] D. F. Brewer and M. J. Nash, "The chinese wall security policy," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1989, pp. 206–214.

[3] S. Akshay, L. Hélouët, and M. Mukund, "Sessions with an unbounded number of agents," HAL-INRIA, Tech. Rep., 2014. [Online]. Available: hal.inria.fr/hal-00979409

[4] A. Finkel and P. Schnoebelen, "Well-structured transition systems everywhere!" *Theor. Comput. Sci.*, vol. 256, no. 1-2, pp. 63–92, 2001.

[5] 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.

[6] J. Misra and W. Cook, "Computation orchestration," *Software and Systems Modeling*, vol. 6, no. 1, pp. 83–110, 2007.

[7] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber, "Active XML: A data-centric perspective on web services," in *BDA02*, 2002.

[8] S. Joshi and B. König, "Applying the graph minor theorem to the verification of graph transformation systems," in *Proceedings of CAV 2008*, ser. LNCS, vol. 5123, 2008, pp. 214–226.

[9] N. Bertrand, G. Delzanno, B. König, A. Sangnier, and J. Stückrath, "On the decidability status of reachability and coverability in graph transformation systems," in *Rewriting Techniques and Applications (RTA'12)*, ser. LIPIcs, vol. 15, 2012, pp. 101–116.

[10] K. Honda, N. Yoshida, and M. Carbone, "Multiparty asynchronous session types," in *POPL*, 2008, pp. 273–284.

[11] P.-M. Deniélou and N. Yoshida, "Dynamic multirole session types," in *POPL*, 2011, pp. 435–446.

[12] R. Demangeon and K. Honda, "Nested protocols in session types," in *CONCUR*, 2012, pp. 272–286.

[13] L. Acciai and M. Boreale, "Deciding safety properties in infinite-state pi-calculus via behavioural types," in *ICALP (2)*, ser. LNCS, vol. 5556, 2009, pp. 31–42.

[14] T. Wies, D. Zufferey, and T. A. Henzinger, "Forward analysis of depth-bounded processes," in *FOSSACS*, ser. LNCS, vol. 6014, 2010, pp. 94–108.

[15] I. Lanese, F. Martins, V. T. Vasconcelos, and A. Ravara, "Disciplining orchestration and conversation in service-oriented computing," in *SEFM*. IEEE Computer Society, 2007, pp. 305–314.

[16] R. Bruni, I. Lanese, H. C. Melgratti, and E. Tuosto, "Multiparty sessions in SOC," in *COORDINATION*, ser. LNCS, vol. 5052, 2008, pp. 67–82.

[17] M. Boreale, R. Bruni, R. De Nicola, and M. Loreti, "Sessions and pipelines for structured service programming," in *FMOODS*, ser. LNCS, vol. 5051, 2008, pp. 19–38.

[18] L. Caires and H. T. Vieira, "Conversation types," *Theor. Comput. Sci.*, vol. 411, no. 51-52, pp. 4399–4440, 2010.

[19] R. Pugliese and F. Tiezzi, "A calculus for orchestration of web services," *J. Applied Logic*, vol. 10, no. 1, pp. 2–31, 2012.

[20] P. Schnoebelen, "Revisiting Ackermann-hardness for lossy counter machines and reset Petri nets," in *MFCS'10 : Mathematical Foundations of Computer Science*, ser. LNCS, vol. 6281, 2010, pp. 616–628.