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 business systems, agents engage in structured interactions, called sessions. Sessions are logical units of computations, like transactions. However, unlike transactions, sessions cannot be isolated from each other. Thus, one has to verify such systems in the presence of both intended and unintended interference between sessions.**

The main 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 has to be modelled without assigning an unbounded set of unique identities to active agents and sessions.

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

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. When multiple transactions execute in parallel, the expected behaviour is specified in terms of notions such as serializability or linearizability that presuppose that transactions are atomic units that do not interact.

Sessions, on the other hand, typically need to interact to achieve the task at hand. Consider a scenario where a customer purchases an item from an online merchant and pays 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—think of customers at an online store. This also allows unboundedly many sessions to be active in parallel. When sessions interact, we need to compare entities across sessions. If we naïvely use unique identifiers for agents and sessions, we have to deal with an unbounded set of names, which again makes verification intractable.

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. The original model only permitted a fixed and finite set of agents. We represent configurations of session systems as labelled graphs. The operational semantics is described 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. Section II introduces session systems, and Section III defines their semantics. Section IV describes how to restrict session systems to well structured transition systems for which coverability is decidable. Section V assembles these results to provide effective verification tools to check conflicts of interest and CWPs. Several formalisms to describe or implement services and their orchestrations have been proposed in the past, and section VI lists some of them and compares with our approach and formalism. The complete operational semantics of session systems and proofs of theorems and important lemmas of the paper are provided in appendix.

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. Agents 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 whenever they need to. 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, ...

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 $true$. We will explain later, in detail, the meaning of each of these transitions.

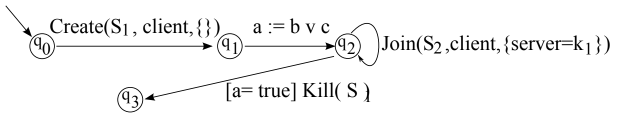


Fig. 1. An example of an archetype

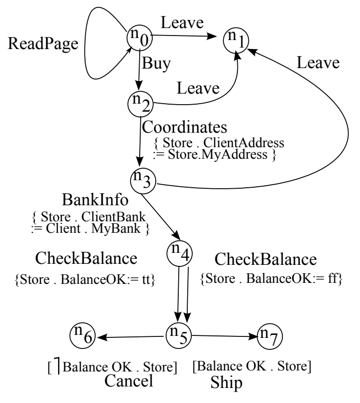
As mentioned earlier, sessions are structured interactions involving a finite set of participating agents. Generic patterns of interactions are called *protocols* and are defined in terms of *roles* (e.g., client, merchant). *Sessions* are instances of these protocols in which roles are instantiated 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 of the same protocol.

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

Figure 2 shows an example of protocol. It represents 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`}. The variables of the system include at least {`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 then checks if the payment is granted by the customer's bank, in which case it 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, for instance, a protocol modelling 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, since we allow shared actions among 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 Fig-



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

Fig. 2. An example of a protocol

ure 2 may seem correct when considering one single session, or when considering that sessions are handled one after the other. However, if several sessions coexist in the system, the balance status *should not* be stored in a global variable by the online merchant. Identifying such problems calls 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 $\mathcal{K}$. The set of *Boolean expressions* over $\mathcal{V} \cup \mathcal{K}$ is defined as follows:

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

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 down to atomic formulas 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 a set of variables $\mathcal{X}$ 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 the expression e evaluates to $true$ 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.

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

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 can only be played by the agent that is referred to as k_j by the agent playing role r_q in a protocol. With this mechanism, an agent can ask to join a session with a particular 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 $blocked$ variable is set to tt. $Kill(s)$ kills all sessions of type s owned by the executing agent in the system.

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, where Σ is the 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 appendix.

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. To ease up 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 $\mathcal{A} = \{A_1, \dots, A_k\}$ is a finite set of archetypes and $\mathcal{S} = \{S_1, \dots, S_q\}$ 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., instanciations of archetypes) and V_S denotes sessions (i.e., instanciations of protocols).
- E is a set of labeled edges over V, representing agent references, and connections between agents and sessions. We have $E \subseteq (V_A \times \mathcal{K} \times V_A) \cup (V_A \times \mathbb{N} \times \{\mathrm{tt}, \mathrm{ff}\} \times V_S) \cup (V_A \times \{qry\} \times V_S)$. A triple $(v, k, v') \in V_A \times \mathcal{K} \times V_A$ represents the fact that the agent A represented by vertex v has the identity of agent A' represented by vertex v' stored via reference variable k. An edge of the form (v, qry, v') means that the agent represented by vertex v is querying the system for a specific kind of session, whose characteristics are described by v'. The number of labels for these two kinds of edges is of course finite. Edges of the form (v, l, s) where $l = (n, b) \in \mathbb{N} \times \{\mathrm{tt}, \mathrm{ff}\}$, $v \in V_A$, and $s \in V_S$ indicates 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 false otherwise.
    - The agent represented by vertex 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 agent that v wishes to join, along with the constraints attached to this request. Here, the flag b is set to ff (v cannot own of 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 at most the number of roles in the protocol it instanciates.
- τ is a labeling of 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 $\tau : V_A \rightarrow \mathcal{A} \times \bigcup_{i \in \mathcal{A}} Q_i \times Val(\mathcal{V})$.
  The labelling $\tau(v) = (A_i, q, val)$ 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 labeling of session vertices. We have $\tau : V_S \rightarrow \mathcal{S} \times \mathcal{N} \times Cnst(\mathcal{S}, \mathcal{K}) \times 2^{\mathcal{R}}$ where $\mathcal{N}$ is the union of all possible nodes appearing in protocols. For every vertex $v \in V_S$ we have $\tau(v) = (s, n, c, J)$ if v represents an instance of a running session. A tuple (s, n, c, J) indicates that vertex v represents 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, we have $\tau(v) = (s, c)$ if v represents a join request or a query for a protocol s with constraints defined by c. In such case, 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.

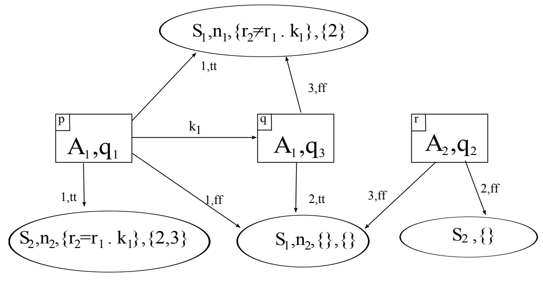


Fig. 3. Graphical representation of configurations

Figure 3 shows a graphical representation of a possible configurations 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 elipses. For clarity, we have not represented variable valuations in the labels of agents. In this configuration, an agent p of (arche)type A_1 is in state q_1, another agent q of type A_1 is in state q_3, and a third agent r of type A_2 is in state q_2. Agent p knows agent q, and has stored its address in its reference variable k_1. Agents p, q and r are involved in a session of type S_1 owned by q, currently in state n_2. Agents p, q, r play roles r_1, r_2, r_3, respectively, within this instance of S_1. 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 r, 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 and r_3, respectively, within the session. Agent p is the owner of this session, and the constraint says that the agent that will join as role r_2 should differ from the agent known by p as entry k_1 in its set of references. Agent r has asked to join an existing session of type S_2, as 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 detailed semantics is available in appendix). 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 change of valuation and state of the agent. *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 variable 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 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 in removing the edge and vertex modeling a query, and changing the querying agent's label to model the change in status of the special variable *blocked*. 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 session systems semantics, we can define the behaviours of a session system as sequences 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 $\rho = C_0 \xrightarrow{\sigma_1} C_1 \dots \xrightarrow{\sigma_k} C_k$. 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, session systems define infinite state transition systems. We refer interested readers to the formal semantics of moves defined in appendix. We will say that a configuration C is *reachable* from an initial configuration C_0, denoted $C_0 \longrightarrow C_k$, if there exists $\rho = C_0 \xrightarrow{\sigma_1} C_1 \dots \xrightarrow{\sigma_n} C$.

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 can be 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 configuration space and as a result verifying even simple reachability properties is undecidable. However, we will now show that under some mild restrictions, these systems are well-structured, which implies that some interesting problems (such as checking coverability) are decidable. In the next section, we will show that this allows to verify interesting properties and in particular business rules such as conflict of interest.

We start with some relevant notions and results from [3]. A *well-quasi ordering* (WQO) on a set X is a reflexive, transitive binary relation $\leq$ such that any infinite sequence $x_0, x_1, \dots$ 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 set* if $\uparrow X = X$. An upward closed set X in a WQO can be represented by a *finite basis* $B(X) = \min_{\leq}\{X\}$. This property is particularly useful: as $X = \cup_{x \in B(X)} \uparrow\{x\}$, one can recall infinite upward closed sets of elements using only a basis as finite set of representatives.

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 $x_1 \geq x$ implies $(x_1, x_1') \in succ^*$ for some $x_1' \geq x'$, where $succ^*$ is the transitive and reflexive closure of relation $succ$. For an upward closed set X, we denote by $pre(X) = \{y \mid \exists x \in X, x \in succ(y)\}$ the set of predecessors of X (by relation $succ$). $pre^*(X) = \{y \mid \exists x \in X, x \in succ^*(y)\}$. The *pred-basis* of X is the finite set $pred_B(X) = B(pre(X))$. We say that a WSTS has *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 [3]):

Proposition 5 (Coverability in WSTS [3]): Let $S = (X, \leq, succ)$ 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 $x' \geq x$.

Let us lift the result of proposition 5 to sessions systems. For this, 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 of a session system. We will say that C_1 is a *subgraph* of C_2, denoted by $C_1 \sqsubseteq C_2$ iff there exists a pair of 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 $l \in \mathcal{K} \cup (\mathbb{N} \times \{tt, ff\}) \cup \{qry\}$, then $\psi'(e) = (\psi(v), l, \psi(v))$ is an edge of E_2.

We will also 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 above. However, $(\mathcal{C}, \sqsubseteq)$ *is not* a WQO: one can design sets of pairwise incomparable configurations of arbitrary sizes. To overcome this problem, we need to restrict the set of configurations considered. One obvious restriction is obtained by setting a bound on the number of agents in configurations. In [1], we have 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 $\langle$session, state, unassigned roles$\rangle$). 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 resource may increase the number of clients a store can handle), and one can not set such 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 will say that a vertex $v_2 \in V_A$ is a successor of vertex $v_1 \in V_A$ iff there exists 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 if 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* of C is a sequence of vertices $v_1, v_2 \dots v_n$ such that v_{i+1} is a successor of v_i in C. This path is simple if $v_i \neq v_j$ for every $i \neq j$, and the length of a simple path is its number of vertices. A configuration C is *k-bounded* if it has no simple paths of length greater than k.

In the rest of the paper, we will denote by $\mathcal{C}$ the set of all configurations, and 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 configuration of $\mathcal{C}^k$ is not bounded, but the way these vertices are connected is constrained.

The only way for a configuration $C \in \mathcal{C}^k$ to evolve into $C' \notin \mathcal{C}^k$ is through a *merge* move by the environment, or by learning the address 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. We are now ready to prove that this restriction to $k-$bounded configurations provides the well-structured property.

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

Proof sketch: We first show that $(\mathcal{C}^k_\top, \sqsubseteq)$ is a WQO, by adapting the result of [4], which shows that labelled undirected graphs of k-bounded path length, ordered by the *induced subgraph* relation is a WQO. Then, the transition relation $\longrightarrow$ is compatible with $\sqsubseteq$, as guards are monotonous w.r.t. $\sqsubseteq$, and as all transitions are finite sequences of graph transformations preserving $\sqsubseteq$. ■

We have proved that $(\mathcal{C}^k_\top, \longrightarrow, \sqsubseteq)$ is a WSTS, and, by definition, $\sqsubseteq$ is effective. But in order to use Proposition 5 to show decidability of coverability for session system, we still need to prove that computation of a pred basis is effective, which we shall do now.

In the rest of the paper, we will use set saturation techniques, and represent upward closed sets as their 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 $\{C_1, C_2, \dots, C_n\} \subseteq \mathcal{C}^k_\top$, the basis of $\bigcup_{i=1}^n \uparrow C_i$ is the set of minimal elements w.r.t. $\sqsubseteq$ in $\{C_1, C_2, \dots, C_n\}$. 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 $B = \{B_1, B_2, \dots, B_n\}$ to decide membership. If X is an upward closed subset of $\mathcal{C}^k_\top$ represented by its finite basis $\{B_1, B_2, \dots, B_n\}$, then checking that $C \in X$ amounts to checking $B_i \sqsubseteq C$ for some B_i in the basis of X.

The usual definitions of *Pre* and upward closure apply to $(\mathcal{C}^k_\top, \longrightarrow, \sqsubseteq)$. For any configuration $C \in \mathcal{C}^k_\top$, we define $Pre(C) = \{C' \mid C' \xrightarrow{\sigma} C\}$. The relation extends to upward closures as follows: $Pre(\uparrow C) = \{C' \mid C' \xrightarrow{\sigma} C'' \sqsupseteq C\}$, and to sets of configurations as follows: for $X \subseteq \mathcal{C}^k_\top$, $Pre(X) = \bigcup_{C \in X} Pre(C)$ and $Pre(\uparrow X) = \bigcup_{C \in X} Pre(\uparrow C)$. Computing a basis for an upward closed set is easy: a basis is defined as the minimal elements of a set w.r.t. $\sqsubseteq$, which guarantees that computing this basis is effective. Computing a *pred-basis* is also effective, and amount to computing the finite set of predecessors of elements in a basis, and then the minimal elements of this set. We then have the following lemma:

Lemma 8: Let X be an upward closed subset of $\mathcal{C}^k_\top$ represented by it basis $\{B_1, B_2, \dots, B_n\}$. 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 $C \sqsubseteq C_k$. The properties of WSTSs, as well as the results of Theorem 7 and Lemma 8 allow us to state the following corollary on decidability of coverability in session systems:

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

Proof: We show that there is an effective algorithm to check, for a given configuration C and a session system S, if there exists a run of S starting from a configuration C_0 that reaches a configuration that subsumes C.

We start from C. We know that $\{C\}$ is a basis for $\uparrow C$. Furthermore, any run that ends in a configuration in $\uparrow C$ is a run that covers C. We are hence looking for a run that starts from C_0 and reaches a configuration in $\uparrow C$. Such a run exists iff $C_0 \in \uparrow pre^*(\uparrow C)$, or equivalently, if $C_b \sqsubseteq C_0$ for some $C_b \in Basis(\uparrow pre^*(\uparrow C))$. As $(\mathcal{C}^k_\top, \longrightarrow, \sqsubseteq)$ is a WSTS (by Theorem 7) with effective $\sqsubseteq$ and effective pred basis computation (by Lemma 8), one can use a set-saturation algorithm to compute a basis for $\uparrow pre^*(\uparrow C)$. The algorithm to compute a basis $PB = Basis(\uparrow pre^*(\uparrow C))$ is a fixpoint algorithm. We start from $PB_0 = Basis(\uparrow C) = \{C\}$, and then compute iteratively $PB^k = PB^{k-1} \cup Basis(\uparrow pre(\uparrow PB^{k-1}))$, and stop as soon as $\uparrow PB^k = \uparrow PB^{k-1}$. It has been proved in [3] that for any upward closed set X, this algorithm is correct and terminates when the pred-basis computation is effective. Hence, at the end of the fixpoint computation, we have $\uparrow \bigcup PB^k = Pre^*(\uparrow C)$.

Now, for any element C_b in PB, there exists a run ρ_b from C_b to a configuration C'_b that covers C. Hence, if $C_b \sqsubseteq C_0$ there exists a run over the same actions than ρ_b from C_0 to a configuration greater than C'_b, and hence greater than C. As $\sqsubseteq$ is effective, it then suffices to build the basis PB, and compare its elements to C_0 to check if C is coverable from C_0. ∎

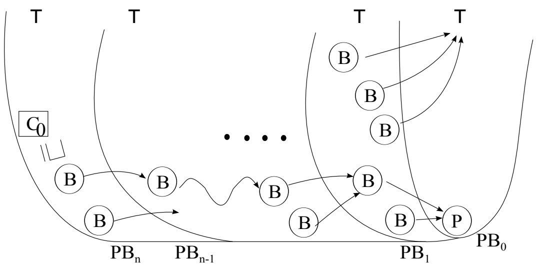


Fig. 4. Computing a Basis for $Pre^*(\uparrow C)$ or $Pre^*(\uparrow P)$

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 huge 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 payment involving the same bank. 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 will say that a configuration C *matches* a pattern P if and only if $P \sqsubseteq C$. Futher, 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 of the run matches P. Finally, a pattern is *reachable* by a session system SS from a configuration C_0 iff 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, basis, 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, a pattern $P \in \mathcal{P}^k$, and a 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: Following the proof of Corollary 9, one can build a finite sequence $PB^0, \ldots, PB^n$ of unions of basis, 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 and whose edge relation is given as follows: there is an edge between an element $x \in (PB^j \setminus PB^{j-1})$ of a basis computed at step j and an element $y \in PB^{j-1}$ computed at step $j-1$ if $\uparrow x \cap Pred(\uparrow y \setminus \top) \neq \emptyset$, i.e., if x is smaller than an element in $Pred(\uparrow y \setminus \top)$. We can then find paths from some $B_0 \in PB^n$ to P in G_{PB}. We will now show that considering these paths suffices to characterize runs over $\mathcal{C}^k$ that meet P.

We start by showing that if there is a path in G_{PB} from some $B_0 \in PB^n$ to P then there is a run over $\mathcal{C}$ from B_0 to a configuration that matches P. For every pair x, y in G_{PB}, and for every configuration $C_x \in \uparrow x$, we know that there exists a configuration C_y and a transition $C_x \longrightarrow C_y$ where $x \sqsubseteq C_x$, $y \sqsubseteq C_y$ and C_y is a successor of C_x. Furthermore, by construction, we have $x, y \neq \top$. Hence, if there exists a path $B_0, B_1, \ldots, B_{k-1}, P$ in G_{PB}, then there exists a run ρ_{B_0} starting from B_0 that meets P. This immediately implies that if $B_0 \sqsubseteq C_0$, there is a run $\rho_{C_0} = C_0 \longrightarrow C_1 \longrightarrow C_2 \longrightarrow \ldots \longrightarrow C_k$ starting from C_0 that meets P, such that $k \leq n$, and every C_i is greater than an element B_i of the graph computed as a temporary basis at step $k - i$.

However, this does not yet imply that the run ρ_{C_0} is over $\mathcal{C}^k$, as the additional edges in $C_0 \setminus B_0$ may force the run to reach $\top$. But now, since $B_0 \sqsubseteq C_0$, there exists an injective mapping from B_0 to C_0, and the number of such injective mappings is finite. For a given mapping γ, we can then compute all consecutive minimal C_i's in the run that modifies only elements that have appeared in the bases computed by the set saturation algorithm. If none of the C_i's occuring along the run meeting P is $\top$, we are done, as we have a witness run from C_0 to a configuration embedding P. On the other hand, if for every $B_0 \sqsubseteq C_0$, every path from B_0 to P in G_{PB} and every way to embed B_0 into C_0, the run towards P encounters a configuration outside $\mathcal{C}^k$, then there is no way to meet P in a run over $\mathcal{C}^k$. Thus, we can also decide if the run ρ_{C_0} in SS corresponding to the path in G_{PB} is in $\mathcal{C}^k$.

Conversely, suppose there exists a run $C_0 \longrightarrow C_1 \longrightarrow \dots C_q$ with $P \sqsubseteq C_q$ over $\mathcal{C}^k$. Hence, necessarily, for every $1 \leq i \leq q$, we have $B_j \sqsubseteq C_i$ for some $B_j \in PB^n$, as PB^n is a basis for all configurations from which a configuration greater than P is accessible. Note that $B_j \sqsubseteq C_i$ does not imply that B_j is computed at step i of the pred-basis computation. However, B_j or a greater basis $B_{j'}$ must have been computed at step $q - max\{i \mid B_j \sqsubseteq C_i\}$ of the set saturation algorithm, i.e., the graph G_{PB} contains a path that goes through basis (not all of them necessarily appear in the final set PB^n) that are successively computed by the set saturation method. This completes the proof, since we have reduced checking existence of a run meeting P to reachability of P in G_{PB}. ■

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 through the whole lifetime of a run of a system, that is, may have to consider several situations that can (or should not) appear along a run. For instance, one wants to guarantee that a client receives an artifact sold by a webstore only after a payment. One may also want to ensure non-competition clauses, that is require that 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 for short [2]). In both examples, the first idea is 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 is however a wrong way of modeling these properties: described this way, P_1 and P_2 may refer to distinct agents and sessions, while the sales or non-competition examples depict situations involving the same participants. To model such situations, we need to mention which elements in both patterns P_1, P_2 refer to the same agents. 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 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 to consider 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 violation of CWPs by a run of a session system over $\mathcal{C}^k$ is a decidable problem. 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 that will help tracing it along a run. This results in a slight change in the semantics, that preserves WQO and compatibility. Then we will 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 $Tg \subseteq 2^T$ is the set of tags used in C, for $v \in V_A, \tau(v) \in L \times T_\perp$, where L is the set of agent and session labels identified in the untagged case, and labels of vertices in V_S and edges in E are defined as in untagged configurations. We extend the ordering $<$ 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 will write $C_1 \sqsubseteq_t C_2$ iff 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, elements of a configuration with distinct tags from T are incomparable, and configurations with uncomparable sets of used tags are uncomparable too.

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 $\to_t$ to take tags into account. The only change is the environment move *create*. While creating a new agent, the create action now randomly fixes a permanent tag t chosen randomly from $T_\perp$ for the new agent, if this tag is not already used, and adds it to the set of tags used. The created 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/disabling actions. 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 $V_S, \tau'(v) = \tau(v)$. 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 iff $P \sqsubseteq UT(C)$. We easily get the following result:

Lemma 14: $(TC^k_\top, \rightarrow_t, \sqsubseteq)$ is a WSTS. Also, if $C \in TC^k_\top$, 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 $\tau_t(v) = (\tau(v), \bot)$ otherwise.
- $P'_t = (V', E', \tau'_t)$ where $\tau'_t(v) = (\tau'(v), t_v)$ if $v \in range(\psi)$ and $\tau'_t(v) = (\tau'(v), \bot)$ 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 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

(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) $C_0 \in Pre^*\left(UT(\uparrow P_t \cap Pre^*(\uparrow P'_t))\right)$ with respect to untagged semantics.

This lemma shows equivalence of correlated pattern matching and of pair of coverability problems, and suffices to proove 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 an effective coverability problem.

VI. Related Work

Several mechanisms and languages have already 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, a purchase order number. 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 that is essential to establish sessions among the 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.

The techniques used for session systems use well quasi ordering on graphs, and set a restriction to get well-structuredness. Within this setting, decidability of coverability is not surprising. Several papers have used graphs transformation systems (GTS) as model and considered verification techniques. [8] studies GTS equipped with the graph minor relation, and show that several subclasses of this model are WSTS. However, the minor relation can not be used in our context to model business rules: a pattern defines actual connections among agents and sessions, and the graph minor, by allowing collapsing of nodes and edges, emphasizes connection among vertices (possibly via paths) rather than 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. Our model does not fall into these subclasses: 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 efforts have also been devoted to services modeling in the π-calculus community. *Session types* [10] have been proposed as a formal model for web services, and have then 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 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 the fragment of well-typed π-calculus processes. The considered fragment can express safety properties. A solution to covering problems for π-calculus with bounded depth have been proposed in [14]. This work shows that for bounded depth π-calculus, a forward coverability algorithm (EEC) terminates, even if the bound is unknown. There are several similarities between configurations that can be reached by bounded depth π-calculus terms and k-bounded configurations (both can be seen as graphs of bounded path length). Note however that boundedness for configurations of session systems is set as an arbitrary restriction to the semantics. So far, we do not know syntactic restrictions to session systems ensuring a bound on paths length in configurations.

Let us now compare sessions systems and π-calculus in terms of modeling power. One can see a sessions as a local name shared by its participants. However, there are some features of session systems that do not find a straightforward translation into π-calculus. First, agents can kill sessions. This feature is essential in many web-based systems, where a server may shut down its activities, and cancel a series of ongoing transactions. In π-calculus, local names can survive the end of a session, and process are not meant to be aborted. Second, joining a session and querying the system to find an occurrence of an existing session are a non-deterministic actions provided by the environment. Though we think that one could model an environment and simulate killing, querying, and joining with 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]. Processes communicate via streams, and choices of a process are implemented as external choices (if-then-else constructs can be implemented this way). This model has an interesting expressive power, as it allows to select values from (ordered) streams, minimal elements, first arrived values, etc, which clearly can not be implemented within our session systems. The counterpart of this expressiveness is of course decidability. Implicitly, processes can interact but run until completion, unlike sessions, that can be interrupted. A multiparty session formalism called $\mu - se$ is proposed by [16]. Its principle is the same as in session systems, that is avoid managing sessions identities explicitly. Sessions in $\mu - se$ allow participating processes to communicate in a private way, and additional communications are allowed among processes that are located on the same site. Arbitrary numbers of sessions can be created on a site. Communications are handled as usual in π-calculus. A merging mechanism allows a process to enter a session at any point, and persistent services can be implemented. Session can handle an arbitrary number of joined processes (while session systems define interactions among sets of agents of predetermined sizes). However, the formalism does not provide means to terminate processes before their completion. The *CASPIS* formalism [17] was influenced by the π-calculus and by ORC, and designed to orchestrate services. It provides pairwise sessions, modeled as service calls which creates private names shared by the caller and callee of a session, and pipelining, i.e. a way for a 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 gives ways to terminate sessions. *Conversation types* [18] is an extension of π-calculus that replaces channel based communications by context sensitive message based communications. A conversation is a behavioral type describing multi party interactions among processes. [18] provides typing mechanisms to ensure that conversations are implemented by processes in a compatible way, and that processes can never get stuck when interleaving sessions. A conversation is close to the notion of session in our setting but allows for unbounded number of participants. Like other other π-calculus variants, this formalism does not allow for abortion of conversations. The *COWS* approach (see for instance [19]) introduces a complete language for the orchestration of services. COWS allows for definition of stateful services, and proposes correlations variables that implement correlations of messages as in BPEL, a wait operation to suspend processes for a chosen time 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 owners of services (and only them) can select the kind of service to be killed.

VII. Conclusion

This paper has presented a session-based formalism to model systems that handle arbitrary numbers of sessions and agents. A restriction of the semantics of the model to runs over bounded path length configurations allows for decision of coverability problems. An immediate consequence is that checking simple properties such as conflict of interest or Chinese Wall Properties for this restriction is also decidable.

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

References

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

[2] David F.C. Brewer and Michael J. Nash, "The chinese wall security policy," in *EEE SYMPOSIUM ON RESEARCH IN SECURITY AND PRIVACY*. IEEE, 1989, pp. 206–214.

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

[4] G. Ding, "Subgraphs and well-quasi-ordering," in *Journal of Graph Theory*, vol. 16(5), 1992, pp. 489 – 502.

[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. [Online]. Available: http://xml.coverpages.org/BPELv11-May052003Final.pdf

[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.