Data Tracking in Parameterized Systems

Giorgio Delzanno[1]

1 **DIBRIS, University of Genova, Italy** giorgio.delzanno@unige.it

Abstract

We study parameterized verification problems for concurrent systems with data enriched with a permission model for invoking remote services. Processes are modelled via register automata. Communication is achieved by rendez-vous with value passing. Permissions are represented as graphs with an additional conflict relation to specify incompatible access rights. The resulting model is inspired by communication architectures underlying operating systems for mobile devices. We consider decision problems involving permission violations and data tracking formulated for an arbitrary number of processes and use reductions to well structured transition systems to obtain decidable fragments of the model.

1998 ACM Subject Classification F1.1 Models of Computation, F1.2 Modes of Computation, D2.4 Software/Program Verification

Keywords and phrases Parameterized verification, well-structured transition systems, constraints, concurrent system with data

1 Introduction

Resource control is a very difficult task in presence of interprocess communication. An interesting example comes from Android applications, whose underlying communication model is based on RPC. In Android processes use special messages, called intents, to start new activities. Intents can contain data that can thus be trasmitted from the caller to the callee. Consider for instance the example in [9]. Assume that processes of type A (Activities in Android) have permissions, statically declared in the Manifest, to retrieve user contact details and to start new instances of process of type B. Furthermore, assume that, upon reception of a start intent, a process of type B extracts the data and send them to a third party. Process interaction does not directly violate the permissions declared in the Manifest. However data exchanged by process instances can lead to leakage of private information.

In this paper we study the problem of resource control from the perspective of parameterized verification, i.e., formal verification of concurrent systems composed by an arbitrary number of components. We model processes as communicating register automata, i.e., automata with a local memory defined by a finite set of registers. Registers are used to store identifiers, an abstract representation of resources. Rendez-vous communication with value passing is used to model remote service invocations and data flows between components. To control access to remote services, we define a permission model using two additional components: a permission graph, whose edges are indicated $A \rightharpoonup B$, where A and B are process types, and a conflict relation $A \not\downarrow B$ again defined on process types.

For instance, we represent the above mentioned example in Fig. 1 where the permission graph contains edges: $C \rightharpoonup A \rightharpoonup B \rightharpoonup I$, C is a process that handles the contents of a device, and I represents a potential intruder. We assume here that $C \not\downarrow I$.

We consider verification problems that capture two types of design errors: permission violations, i.e., a process instance invokes a service without rights, and conflict detection, i.e., during a computation an identifier is transferred from a process of type A to a process

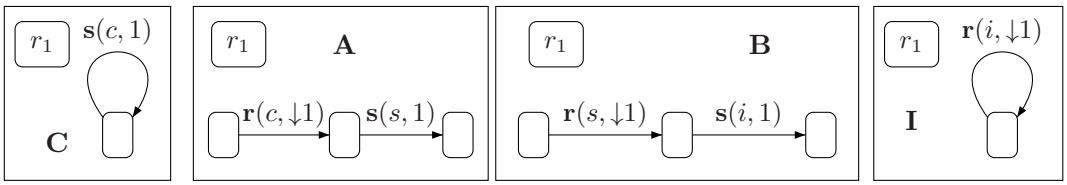

Figure 1 Register automata for our example.

of type B such that A and B are incompatible. Since processes are designed to operate in an open environment, it is particularly interesting to consider verification problems in which the number of concurrent processes in the initial configuration is not fixed a priori, i.e., parameterized verification. In the paper we first show that, despite of the fact that permission graphs are statically defined on finitely many process types, parameterized verification of permission violations and conflict detection is undecidable in general. We then consider fragments of the model by restricting the interplay between registers and message fields, and show that it is possible to obtain non trivial fragments for which we can give decision procedures for both properties. For properties that require data tracking, the proof consists of two steps: we first extend the operational semantics with predicates that encode footprints of data exchanged by processes. The additional information represent an unbounded memory containing footprints that share information with the current state. The alphabet used to represent the memory is infinite. We then show that the resulting semantics can be represented symbolically via a low level language based on rewriting and constraints [11] and infer decidability results from those obtained in that setting. In the general case of $r > 1$ registers, the link with rewriting and constraints can be exploited to apply a symbolic backward reachability engine [10] as a possibly non terminating procedure to verify absence of violations and conflicts for any number of component instances.

2 Process Model

We model a concurrent system using a collection of interacting processes. Each process is described by an automaton with operations over a finite set of registers. Data are identifiers viewed as handlers to more complex resources. Communication is achieved via rendez-vous, an abstraction of synchronization, message passing and remote procedure calls. We assume here that send and receptions are executed without delays. A process can transmit part of its current data to other nodes by inserting the current value of some of its registers inside messages. Messages carry both a type and a finite tuple of data. A receiver can either compare the data contained inside messages with the current values of its registers, can store data in the registers or simply ignore some of the fields in the message payload.

Let us first describe the set of actions. We use $r \geq 0$ and $f \geq 0$ to denote resp. the number of registers in each node and the number of data fields available in each message and consider a finite alphabet Σ of message types.

The set of actions $\mathcal{A}$ is defined as follows. Local actions are defined by labels $\mathbf{l}(m)$ where $m \in \Sigma$. Send actions are defined by labels $\mathbf{s}(m, \overline{p})$, where $m \in \Sigma$, $\overline{p} = p_1, \ldots, p_f$ and $p_i \in [1..r]$ for $i \in [1..f]$. The action $\mathbf{s}(m, \overline{p})$ corresponds to a message or remote procedure call of type m whose i-th field contains the value of the register p_i of the sending node. For instance, in $\mathbf{s}(req, 1, 1)$ the current value of the register 1 of the sender is copied in the first two fields of the message.

The set of field actions Op^r is defined as $\{?k, \downarrow k, * \mid k \in [1..r]\}$. When used at position i of a reception action, $?k$ tests whether the content of the k-th register is equal to the i-th field of the message, $\downarrow k$ is used to store the field into the k-th register, and $*$ is used to ignore the field. Reception actions are defined by labels $\mathbf{r}(m, \overline{\alpha})$, $\overline{\alpha} = \alpha_1, \ldots, \alpha_f$, where $m \in \Sigma$, $\alpha_i \in Op^r$ for $i \in [1..f]$.

As an example, for $r = 2$ and $f = 3$, $\mathbf{r}(req, ?2, *, \downarrow 1)$ specifies the reception of a message of type req in which the first field is tested for equality against the current value of the second register, the second field is ignored, and the third field is assigned to the first register.

▶ **Definition 1.** A *process definition* over Σ is a tuple $\mathcal{D} = \langle Q, R, q_0 \rangle$ where: Q is a finite set of control states, $q_0 \in Q$ is an initial control state, and $R \subseteq Q \times \mathcal{A} \times Q$.

We use $\mathcal{D} = \{D_1, \ldots, D_n\}$ to denote a set of process definitions such that $D_i = \langle Q_i, R_i, q_0^i \rangle$, and $\mathcal{Q} = Q_1 \cup \ldots \cup Q_n \cup \{error\}$ to denote the set of all control states. We assume here that $Q_i \cap Q_j = \emptyset$ and that $error \notin Q_i$ for all i, j.

In the rest of the paper, we will use definitions as process types, i.e., we will say that a process has type D if its behaviour is defined by the automata D.

▶ **Definition 2.** Process Definitions with Permissions (PDP) are defined by a graph $\mathcal{G} = (\mathcal{D}, \rightharpoonup)$, where $\mathcal{D}$ is a set of process definitions, and $\rightharpoonup \subseteq \mathcal{D} \times \mathcal{D}$ is a set of permission edges. $D_1 \rightharpoonup D_2$ is used to denote $\langle D_1, D_2 \rangle \in \rightharpoonup$.

The permission graph defines a dependency relation between process definitions. Namely, if $D_1 \rightharpoonup D_2$, then a process of type D_1 has the permission to use services provided by a process of type D_2.

2.1 Operational Semantics

We now move to the definition of an operational semantics for our model. First of all, values of registers are taken from a denumerable set of identifiers Id. A configuration γ is a tuple $\langle V, L \rangle$, where $V = \{n_1, \ldots, n_k\}$ is a set of process instances for $k \geq 0$, $L : V \to \mathcal{D} \times \mathcal{Q} \times Id^r$ is a labeling function that associates a definition, a control state (taken from the union of control states of all definitions), and values to registers of each process. We use Γ to denote the infinite set of all configurations (the set is infinite since it contains configurations with any number of process instances).

Terminology For a process $v \in V$, we denote by $L_D(v)$, $L_Q(v)$ and $L_M(v)$ the three projections of $L(v)$. With an abuse of a notation, we use the same notation to extract the projections relative to a given node v from a configuration γ, i.e., $L_D(\gamma, v) = L_D(v)$ is the definition (or type) associated to node v in γ; $L_Q(\gamma, v) = L_Q(v)$ is the current state of node v in γ; and $L_M(\gamma, v, i) = L_M(v)[i]$ is the current value of register i of node v in γ. Finally, the configuration γ is said to be initial if (1) all nodes are in their initial control states, i.e., for all $v \in V$, $L_Q(v) = q_0$ if $L_D(v) = \langle Q, R, q_0 \rangle$; (2) for all nodes, all registers contain different values, i.e., for all $u, v \in V$ and all $i, j \in [1..r]$, if $u \neq v$ or $i \neq j$ then $L_M(v)[i] \neq L_M(v)[j]$. We use $\Gamma_0 \subseteq \Gamma$ to denote the infinite subset of initial configurations.

For a configuration $\gamma = \langle V, L\rangle$, $u, v \in V$, $\overline{p} = p_1, \ldots, p_f$ and an action $A = \mathbf{s}(m, \overline{p})$, let $S(v, u, A) \subseteq \mathcal{Q} \times Id^r$ be the set of the possible labels that can take u on reception of the message m sent by v, i.e., we have $(q', M) \in S(v, u, A)$, where M is an r-tuple of identifiers, if and only if there exists a receive action of the form $\langle L_Q(u), \mathbf{r}(m, \overline{\alpha}), q'\rangle$ where $\overline{\alpha} = \alpha_1, \ldots, \alpha_f$ verifying the following condition: For all $i \in [1..f]$, (1) if $\alpha_i = ?j$, then $L_M(u)[j] = L_M(v)[p_i]$; (2) if $\alpha_i = \downarrow j$ then $M[j] = L_M(v)[p_i]$, otherwise $M[j] = L_M(u)[j]$.

Given $\mathcal{G} = \langle \mathcal{D}, \rightharpoonup\rangle$, we define the transition system $TS_{\mathcal{G}} = \langle \Gamma, \Rightarrow\rangle$, where $\Rightarrow \subseteq \Gamma \times \Gamma$. Specifically, for $\gamma = \langle V, L\rangle$ and $\gamma' = \langle V, L'\rangle \in \Gamma$, $A = \mathbf{s}(m, \overline{p})$ we have $\gamma \Rightarrow \gamma'$ if and only if (1) for all $u \in V$, $L_D(\gamma', u) = L_D(\gamma, u)$, and (2) one of the following conditions holds:

- there exist $u, v \in V$ $u \neq v$ s.t. $\langle L_Q(\gamma, v), A, L_Q(\gamma', v)\rangle \in R$, $L_M(\gamma', v) = L_M(\gamma, v)$ $\langle L_Q(\gamma', u), L_M(\gamma', u)\rangle \in S(v, u, A)$, $L_D(\gamma, v) \rightharpoonup L_D(\gamma, u)$, $L_Q(\gamma', u') = L_Q(\gamma, u')$ and $L_M(\gamma', u') = L_M(\gamma, u')$ for $u' \in V$ s.t. $u' \neq u, v$.
- there exist $v, u \in V$ $v \neq u$, $q_1, q_2 \in \mathcal{Q}$, and $M \in Id^r$ s.t. $\langle L_Q(\gamma, v), A, q_1\rangle \in R$, $\langle q_2, M\rangle \in S(v, u, A)$, $L_D(\gamma, v) \not\rightharpoonup L_D(\gamma, u)$, $L_Q(\gamma', v) = error$, $L_M(\gamma', v) = L_M(\gamma, v)$, $L_Q(\gamma', u') = L_Q(\gamma, u')$, $L_M(\gamma', u') = L_M(\gamma, u')$, for $u' \in V$ (including u) s.t. $u' \neq u$.
- there exist $v \in V$, $\langle L_Q(\gamma, v), \mathbf{l}(a), L_Q(\gamma', v)\rangle \in R$ and $L_Q(\gamma', u) = L_Q(\gamma, u)$, $L_M(\gamma', u) = L_M(\gamma, u)$ for $u \in V$ s.t. $u \neq v$.

We use $\stackrel{*}{\Rightarrow}$ to denote the reflexive and transitive closure of $\Rightarrow$.

Finally, given $\mathcal{G} = \langle \mathcal{D}, \rightharpoonup\rangle$ with $TS_{\mathcal{G}} = \langle \Gamma, \Rightarrow\rangle$, the set of reachable configurations is defined as follows: $\mathit{Reach}(\mathcal{G}) = \{\gamma \in \Gamma \mid \exists \gamma_0 \in \Gamma_0 \text{ s.t. } \gamma_0 \stackrel{*}{\Rightarrow} \gamma\}$. We observe that the number of nodes in V does not change during a computation, i.e., all successors of a given configuration γ_0 have the same set of nodes V. However, assuming that $\mathcal{D} \neq \emptyset$, the set of initial configuration Γ_0 is infinite by construction and contains all possible combinations (of any number) of instances of process with types in $\mathcal{D}$. Therefore, $\mathit{Reach}(\mathcal{G})$ is always infinite when $\mathcal{D} \neq \emptyset$.

Detection of Permission Violations

Given a PDP $\mathcal{G} = \langle \mathcal{D}, \rightharpoonup\rangle$, our goal is to decide whether there exists an initial configuration containing any number of process instances of any type from which it is possible to reach a configuration exposing a permission violation, i.e., containing a process with *error* control state. The formal definition of the decision problem is given below.

▶ **Definition 3.** Given a PDP $\mathcal{G} = \langle \mathcal{D}, \rightharpoonup\rangle$ s.t. $\mathcal{D} \neq \emptyset$, with $TS_{\mathcal{G}} = \langle \Gamma, \Rightarrow\rangle$, the problem $\mathit{VD}(r, f)$ is defined as follows: $\exists \gamma \in \mathit{Reach}(\mathcal{G})$ with nodes in V and $\exists v \in V$ such that $L_Q(\gamma, v) = error$?

As remarked in the previous section, Γ_0 is an infinite set of configurations. Hence for fixed r, f $\mathit{VD}(r, f)$ cannot be solved directly by using a reduction to a finite-state system. Intuitively, we need to guess an adequate number of processes in the initial configuration to expose a violation. We will show that in general this is not possible in algorithmic way.

Data Tracking

We are also interested in tracking data exchanged by different processes during a computation and data can generate violations of permissions that are invisible to the $\rightharpoonup$ dependency relation. More specifically, we first introduce a symmetric relation $\lightning \subseteq \mathcal{D} \times \mathcal{D}$ to specify (a priori) potential conflicts between permissions associated to process types (i.e. definitions in $\mathcal{D}$). We now consider the extended model PDP with conflicts (PDPC), defined as $\mathcal{G} = \langle \mathcal{D}, \rightharpoonup$

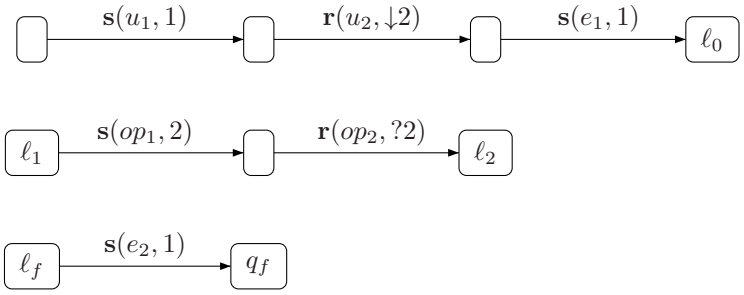

Figure 2 Process of type C: we use op_1 and op_2 to denote messages needed for completing an entire simulation step of instruction op; op is a label in $\{incN, zeroN, nzeroN, decN\}$ for a counter N, .

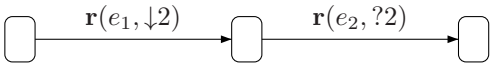

Figure 3 Component of type E.

$, \not\downarrow\rangle$. For instance, if processes of type D_1, D_2 can access internet services and processes of type D_3 cannot, then we assume that $D_1 \not\downarrow D_3$ and $D_2 \not\downarrow D_3$.

We now move to the second decision problem that we consider in the paper.

▶ **Definition 4.** Given PDPC $\mathcal{G} = \langle \mathcal{D}, \rightharpoonup, \not\downarrow \rangle$ with $TS_{\mathcal{G}} = \langle \Gamma, \Rightarrow \rangle$, the problem $CD(r, f)$ is defined as follows: $\exists \gamma_1, \gamma_2 \in Reach(\mathcal{G})$ with nodes in V, $\exists u, v \in V$, and $\exists$ registers i, j such that $\gamma_1 \stackrel{*}{\Rightarrow} \gamma_2$, $L_M(\gamma_1, u, i) = L_M(\gamma_2, v, j)$, and $L_D(\gamma_1, u) \not\downarrow L_D(\gamma_2, v)$?

Finally, we say that a PDP [resp. a PDPC] is violation-free [resp. conflict-free] if and only if there are no violations of the above mentioned types. As for VD, in the CD decision problem there are no restrictions a priori on the number of component instances in the initial configuration. From a computational perspective, this feature is a major obstacle for algorithmic solutions to the problem.

3 Violation Detection

In this section we prove that violation detection is undecidable for $r \geq 2, f = 1$. This property is due to the special parameterized formulation of the problem. The possibility of choosing an initial configuration of arbitrary size can be exploited to set up a network configuration in which a special node plays the role of controller linked to a finite but arbitrary sequence of nodes that encode unitary elements of a memory (e.g. a counter or the tape of a Turing machine). Elements of the memory are linked via identifiers stored in registers. By setting up a specific set of process definitions and an adequate permission relation, it is possible to reduce the halting problem of a counter machine to violation detection. The statement is proved formally in the rest of the section.

▶ **Theorem 5.** *The $VD(2, 1)$ problem is undecidable.*

We exhibit a reduction from the termination problem for two counter machines. Counter machines are sequential programs that manipulate a finite set of counters with values over natural numbers. We consider here instructions such as inc_i, dec_i, $if\ zero_i\ goto_j$, $if\ notzero_i\ goto_j$ for $i \in [1, \ldots, r]$ (number of counters) and $j \in [1 \ldots k]$ (instructions) and

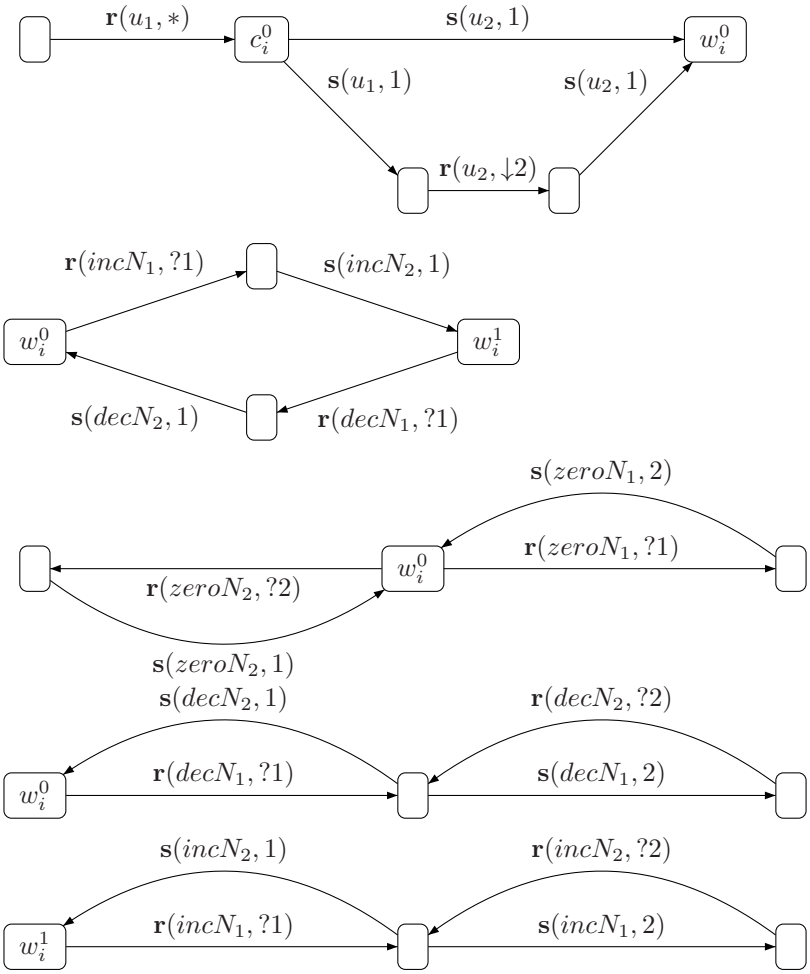

Figure 4 Process of type U.

programs P with instructions $I_1, \ldots, I_k$. For the encoding, we need process definitions C, U and E whose permission graph is as follows: $C \rightharpoonup U, U \rightharpoonup C, C \rightharpoonup E$. An instance I_C of process definition C is used to keep track of the current instruction of P (program counter). Furthermore, in the initialization phase I_C has the following tasks: synchronization with a process of type E used in the last step of the simulation, construction of a linked list, connected to I_C whose elements are instances of type U. Processes of type C and U have two registers, *id* and *next* for simplicity. Register *id* is used as identifier of each process instance. Register *next* is used as pointer to the first/next process instance (the next cell in the list). Instances of type U simulate the unit of a counter c_i, its state denotes a zero or one value for c_i. The types of the elements in the list are chosen non-deterministically. In other words we represent the current values of all counters in a single list.

To create a list of finite but arbitrary length, we just need to first propagate a request message through U cells. U cells can non-deterministically decide to stop propagation and return their identifier to a process of type C. In this phase, upon reception, a process instance stores the identifier in the second register, the “next pointer" and sends its own identifier to another instance. Several lists can be constructed in parallel starting from different initial states. The acknowledgment phases is then needed to build a well formed list in which each node has the identifier of the next cell. Observe that, due to the non-

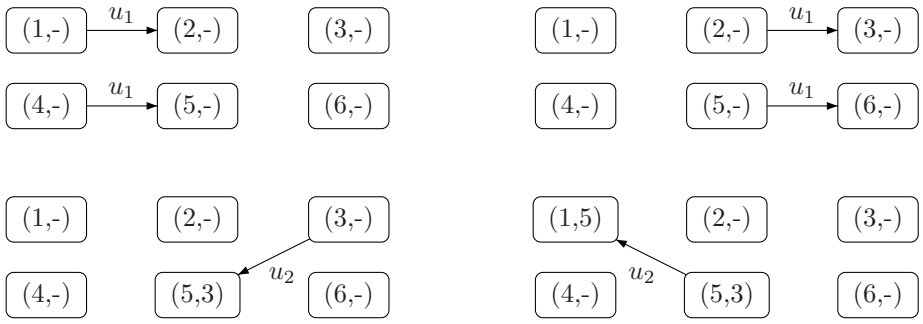

Figure 5 Example of list construction in which the second phase is used to fix the "next" pointers.

determinism of rendez-vous, a well formed lists can have elements taken from other lists constructed in the first phase. An alternative algorithm can be obtained by constructing a list backwards, i.e., propagating the "next pointer" sending id to a node that directly stores in its second register. We adopt the first algorithm so as to use process of type C as initiator and coordinator of all the phases and to show the power of data to isolate special topologies in a fully connected set of processes. When the list is ready, I_C synchronizes, via handshaking, with one instance I_E of a process of type E. The simulation of the program P can now start. The list denotes value k for counter c_i if it contains k instances of process U with an internal state that encode a single unit for the counter c_i. Process U is such that, upon reception of an operation request, it can either execute it locally (e.g. increment of a zero cell) or forward the request to the next cell and wait for an answer (in state w). Simulation of an increment on counter C_i propagates the request to set to one the internal state of a cell of type c_i with value zero in the list. Simulation of a decrement on counter C_i propagates the request to set to zero the internal state of a cell of type c_i with value one in the list. Simulation of a zero-test on counter C_i propagates the test on on the value of the cell to the whole list. An acknowledgment is sent back to the sender if all cells are zero. For a non-zero test the first non-zero cell sends an acknowledgement back to the sender.

The last phase of the simulation starts when the simulation of the counter machine terminates, i.e., I_C has a control state that corresponds to the halt location of P. I_C then sends a special request pv to the first U cell in the list. Upon reception of the pv request, the cell tries to call an action of the I_E instance, generating a permission violation. The definition of process C, E and U are given in Fig. 2, 3, and 4, respectively.

By construction, the counter machine P terminates if and only if there exists an initial configuration from which we can generate a configuration with well-formed lists, and enough memory cells, that can simulate a complete execution of the program P. Formally, P reaches location ℓ_f if an only if there exists an initial configuration γ_0 s.t. $\gamma_0 \Rightarrow^* \gamma_1$ and $L_Q(\gamma_1, u) = \ell_f$ for some node u. From the previous property and following from the interaction between C and E processes, it follows that P reaches location ℓ_f if and only if there exists an initial configuration γ_0 s.t. $\gamma_0 \Rightarrow^* \gamma_1$ and $L_Q(\gamma_1, u) = error$ for some node u. Therefore halting of P is reduced to violation detection in the application $\mathcal{D} = \langle C, E, U \rangle$.

When processes do not exchange data, i.e., $r = 0$ or $f = 0$, it is possible to decide violation detection by using algorithms for deciding the coverability problem in Petri Nets (see appendix for main definitions). Formally, the following property holds.

▶ **Theorem 6.** *The VD(r, f) problem is decidable if either $r = 0$ or $f = 0$.*

Proof. Let $\mathcal{D} = \{C_1, \ldots, C_n\}$ with $C_i = \langle Q_i, \Sigma_i, \delta_i, q_0^i \rangle$ for $i : 1, \ldots, n$. The reduction is defined as follows. The set P of places is defined as $P = \{err\} \cup (\bigcup_{i=1}^{n} Q_i)$. The transitions

are defined as follows. For $i \in \{n\}$ and every rule $r = \langle q, a, q' \rangle \in \delta_i$, we define a transition $t_r = \langle Pre, Post \rangle$ s.t. $Pre = \{q\}$ and $Post = \{q'\}$. For $i, j \in \{n\}$ and every pair of rules $r_1 = \langle q_1, \mathbf{s}(a), q_1' \rangle \in \delta_i$, $r_2 = \langle q_2, \mathbf{r}(a), q_2' \rangle \in \delta_j$ s.t. $C_i \rightharpoonup C_j$, we define a transition $t_r = \langle Pre, Post \rangle$ s.t. $Pre = \{q_1, q_2\}$ and $Post = \{q_1', q_2'\}$. For $i, j \in \{n\}$ and every pair of rules $r_1 = \langle q_1, \mathbf{s}(a), q_1' \rangle \in \delta_i$, $r_2 = \langle q_2, \mathbf{r}(a), q_2' \rangle \in \delta_j$ s.t. $C_i \not\rightharpoonup C_j$, we define a transition $t_r = \langle Pre, Post \rangle$ s.t. $Pre = \{q_1, q_2\}$ and $Post = \{err, q_2\}$.

Given a configuration $\gamma = \langle q_1, \ldots, q_n \rangle$ we define the associated marking M_γ that contains as many occurrences of state q as those in γ. By construction of our reduction, we obtain that γ_0 reaches configuration γ_1 iff $M_{\gamma_0} \triangleright M_{\gamma_1}$. To represent an arbitrary initial configuration, we add a transition that non-deterministically adds token to places that represent initial states of processes. Furthermore, to separate initialization and normal operations we can simply use and additional place *ok* and use it to block process interactions during the initialization phase. Namely, $it_i = \langle Pre, Post \rangle$ s.t. $Pre = \{init\}$ and $Post = \{init, q_0^i\}$ for any i. Furthermore, $it_{i+1} = \langle Pre, Post \rangle$ s.t. $Pre = \{init\}$ and $Post = \{ok\}$. All the other rules are modified in order to add *ok* to their pre-set. As a corollary, we have that the safety property holds for $\mathcal{D}$ iff coverability holds for the Petri net P and the two markings $M_{\gamma_0} = \{init\}$ and $M_{err} = \{e\}$. Following from properties of Petri Nets, we obtain that the safety problem is decidable for $VD(0, 0)$. ◀

In the rest of the paper we will focus our attention on conflict detection and derive other decidability results for violation detection as a side-effect of more general results obtained for processes and messages with data.

4 Conflict Detection

In this section we move to the analysis of the conflict detection problem. We first observe that conflict detection is undecidable for $r > 1$. The proof is similar to the encoding of counter machines used for the undecidability of violation detection. Instead of generating a permission violation as a last step of the simulation the controller C sends its identifier to a special process of type D s.t. C and D are in conflict. This way, a conflict is detected starting from some initial configuration if and only if the counter machine program terminates. We show that both violation and conflict detection are decidable for $r = f = 1$.

The decidability proof consists of two steps. We first extend the transition system of a PDPC by adding a sort of external memory in which to keep track of footprints of identifiers. Footprints are represented via a collection of predicates that mark all types of processes in which an identifier has been stored. It is important to remark that footprints share data with current configurations, i.e., the alphabet used to define footprints is infinite as for configurations. To deal with the conflict detection problem, we may need infinite set of footprints since the problem is parametric in the initial configuration. By extending the transition relation with historical information we can reduce conflict detection to a reachability problem formulated over the predicates in the history. It is important to notice that for this kind of problem we just need a monotonically increasing external memory. The second step of the proof consists in reducing the reachability problem for the transition system with history to coverability in a formalism called MSR(C) that is a special class of multiset rewriting with constraints. The desired result follows then by observing that, for $r = 1$, the resulting encoding produces only rewriting rules with monadic predicates. We can then apply the decision procedure based on the theory of well-structured transition systems defined in [11] to solve algorithmically the CD problem. We start from defining the extended operational semantics.

4.1 Transition System with History

Given $\mathcal{D}$, let us consider the set of unary predicates $P_{\mathcal{D}} = \{h_C | C \in \mathcal{D}\}$. We use the formula $h_C(id)$ to denote a footprint for the identifier $id \in Id$. Let $F_{\mathcal{D}} = \{h_C(id) | id \in Id,\ h_C \in P\}$ be the set of all footprints associated to $\mathcal{D}$. We use $H_{\mathcal{D}}$ to denote all possible multisets of footprints in $F_{\mathcal{D}}$, i.e., $H_{\mathcal{D}} = F_{\mathcal{D}}^{\oplus}$. For a configuration γ, let $fp(\gamma)$ be the multiset of footprints such that $h_C(id) \in fp(\gamma)$ iff the identifier id occurs in some register of a process of type C in γ. As an example, for γ with nodes n_1, n_2, n_3 of type A, B, C with values in the two registers resp. $(1,2)$, $(2,3)$, and $(3,4)$, $fp(\gamma) = \{h_A(1), h_A(2), h_B(2), h_B(3), h_C(3), h_C(4)\}$.

Extended configurations are tuples of the form $\gamma[h]$, where γ is a configuration and $h \in H_{\mathcal{D}}$. An initial configuration is defined then as $\gamma_0[h_0]$, where $h_0 = fp(\gamma_0)$, i.e., h_0 contains the footprints for each identifier in γ_0.

The extended transition relation $\Rightarrow_h \subseteq \Gamma \times H_{\mathcal{D}}$ is built on top of $\Rightarrow$ as follows:

- $\gamma[h] \Rightarrow_h \gamma'[h]$ if $\gamma \Rightarrow \gamma'$ via an application of a local rule;
- $\gamma[h] \Rightarrow_h \gamma'[h \oplus fps(\gamma')]$ if $\gamma \Rightarrow \gamma'$ via an application of a rendez-vous step, where $fps(\gamma')$ is the multiset of footprints in $fp(\gamma')$ generated by store operations after a reception, i.e., $h_C(id) \in fps(\gamma')$ iff $h_C(id) \in fp(\gamma')$ and id is stored in a register of a process of type C during the rendez-vous, for some C and identifier id.

We use $\stackrel{*}{\Rightarrow}_h$ to denote the reflexive and transitive closure of $\Rightarrow_h$.

The CD problem can now be formulated by considering the history of a computation. Namely, CD amounts to checking whether there exists an initial configuration $\gamma_0[h_0]$ (of arbitrary shape) from which it is possible to reach a configuration $\gamma[h]$ such that h contains at least two footprints $h_C(i)$ and $h_D(i)$ for some process type C, D s.t. $C \not\downarrow D$. It is important to observe that we just need unary predicates to represent footprints for identifiers.

4.2 Encoding into MSR(Id)

An encoding of the problem into coverability of a model inspired by Petri nets with identifiers called MSR(C) in which the constraint system consists of equalities. In the encoding we model both the behavior of an application as well as footprints of data exchanged by instances of processes. Footprints are represented via monadic predicates. Each predicate keeps track of the history of a given identifier during its lifetime, namely every type of process in which the identifier has been stored during execution. The history can then be queried in search for conflicts. MSR($\mathcal{C}$) is a formal model for concurrent systems based on a combination of rewriting and constraints. A constraint system $\mathcal{C}$ is defined by formulas with free variables in V, an interpretation domain $\mathcal{D}$, and a satisfiability relation $\models$ for formulas in $\mathcal{C}$ interpreted over $\mathcal{D}$. We use $\mathcal{D} \models_\sigma \varphi$ to denote satisfiability of φ via a substitution $\sigma : Var(\varphi) \to \mathcal{D}$, where $Var(\varphi)$ is the set of free variables in φ. For a fixed set of predicates P, an atomic formula with variables has the form $p(x_1, \ldots, x_n)$ where $p \in P$ and $x_1, \ldots, x_n \in V$. A rewriting rule has the form $M \to M' : \varphi$, where M and M' are multiset of atomic formulas with variables over P and V, and φ is a constraint formula over variables $Var(M \oplus M')$ occurring in $M \oplus M'$. We use $M = A_1, \ldots, A_n$ to denote a multiset of atoms.

MSR(Id) is the instance obtained by considering the constraint system Id defined as follows. Constraint formulas are defined by the grammar $\varphi ::= \varphi_1, \varphi_2 | x = y | x < y$ for variables $x, y \in V$. Here φ_1, φ_2 denotes a conjunction of formulas φ_1 and φ_2. The interpretation domain is defined over an infinite and ordered set of identifiers $\langle Id, =, < \rangle$. For substitution $\sigma : V \to Id$, $x = y$ is interpreted as $\sigma(x) = \sigma(y)$, $x < y$ is interpreted as $\sigma(x) < \sigma(y)$, and φ_1, φ_2 is interpreted as $\sigma(\varphi_1) \wedge \sigma(\varphi_2)$. A constraint φ is satisfied by a substitution σ if $\sigma(\varphi)$ evaluates to $true$. An instance $M\sigma \to M'\sigma$ of a rule $M \to M' : \varphi$ is defined by taking a

substitution $\sigma : Var(M \oplus M') \to Id$ such that $\sigma(\varphi)$ is satisfied in the interpretation Id.

As an example, consider the rule $p(x,y), q(x) \to p(x,y), q(x), q(u) : x < u$ The intuition is that processes $p(x,y)$ and $q(z)$ synchronize when $x = z$ and generate a new instance $q(u)$ with $x < u$. By associating natural numbers to identifiers, $p(1,2), q(1) \to p(1,2), q(1), q(4)$ and $p(3,10), q(3) \to p(3,10), q(3), q(8)$ are two instances of the considered rule. We use $Inst(\Delta)$ to indicate the infinite set of instances of a set Δ of MSR rules.

A configuration is a multiset N of atoms of the form $p(d_1, \ldots, d_n)$ with $d_i \in Id$ for $i : 1, \ldots, n$. For a set Δ of rules and a configuration N, a rewriting step is defined by the relation $\triangleright$ s.t. $N = (M \oplus Q) \triangleright (M' \oplus Q) = N'$ for $(M \to M') \in Inst(\Delta)$. A computation is a sequence of configurations $N_1 \ldots N_m \ldots$ s.t. $N_i \triangleright N_{i+1}$ for $i \geq 0$. For a set of rules Δ, an initial configuration N_0, and a zerary predicate p_f, the coverability problem, COV, consists in checking whether there exists a computation from N_0 to a configuration N_1 s.t. $p_f \in N_1$.

We now encode the $CD(r,f)$ problem as a coverability instance in monadic MSR(Id). Let $\mathcal{D} = \{C_1, \ldots, C_n\}$ with $C_i = \langle Q_i, R_i, q_0^i \rangle$ for $i : 1, \ldots, n$. We define the set $P = \{init, max, ok\} \cup \{h_C | C \in \mathcal{D}\} \cup (\bigcup_{i=1}^{n} Q_i)$ of monadic predicates. Predicates $init$ and ok are used to separate the initialization phase from the simulation steps. Predicates h_C define footprints for data. In order to represent an arbitrary initial configuration we define MSR(Id) rules that non-deterministically add predicates with distinct identifiers. The rules are defined as follows:

- $init \to init, max(x) : true$ where max is used to generate fresh identifiers;
- for each initial state q_0 of process $C \in \mathcal{D}$ with $\overline{x} = x_1, \ldots, x_n$,

$$init, max(x) \to \ init, max(y), q_0(\overline{x}), h_C(x_1), \ldots, h_C(x_n) : y > x_1 > x_2 \ldots x_n > x$$

  injects a node with initial state q_0 and registers initialized to fresh values $x_1, \ldots, x_n$. The value stored in max is reset to a value greater than the last value seen so far.
- The initialization is non-deterministically terminated by the rule $init, max(x) \to ok : true$. Inserting ok marks the beginning of the simulation phase.

The simulation of process rules is defined by the following rules.

Local For $i \in \{n\}$ and every rule $r = \langle q, \mathbf{l}(a), q' \rangle \in R_i$, we define a rewriting rule

$$ok, q(x_1, \ldots, x_n) \to ok, q'(x'_1, \ldots, x'_n) : x'_1 = x_1, \ldots, x'_n = x_n$$

where $x_1, \ldots, x_n$ denote the current values of the registers, and q/q' is the current/next state.

Rendez-vous For $i, j \in \{n\}$ and every pair of rules $r_1 = \langle q_1, \mathbf{s}(a, p_1, \ldots, p_f), q'_1 \rangle \in R_i$, $r_2 = \langle q_2, \mathbf{r}(a, \alpha_1, \ldots, \alpha_f), q'_2 \rangle \in R_j$ s.t. q_1 is the state of a process of type A, q_2 is the state of a process of type B, $A \rightharpoonup B$, we define a rule

$$ok, q_1(x_1, \ldots, x_r), q_2(y_1, \ldots, y_r) \to ok, q'_1(x'_1, \ldots, x'_r), q'_2(y'_1, \ldots, y'_r) \oplus M_f : \varphi$$

where φ is the constraint $\varphi_1, \ldots, \varphi_f, \psi_1, \psi_2$ defined as follows: for $i : 1, \ldots, f$, if $\alpha_i = ?p_j$, then φ_i is the equality $y_i = x_j$ (each guard must be satisfied); if $\alpha_i = \downarrow p_j$, then φ_i is the equality $y'_i = x_j$ (assignment to register i) and M_f contains predicate $h_B(x_j)$ encoding a footprint; if $\alpha_i = *$, then $\varphi_i = true$. Furthermore, $\psi = (\wedge_{i=1}^{r} x'_i = x_i) \wedge (\wedge_{i=1, \alpha_i \neq \downarrow k, k \geq 1}^{r} y'_i = y_i)$ to denote that values of registers remain unchanged unless modified by some store operations in the receiver process.

Violation For $i, j \in \{n\}$ and every pair of rules $r_1 = \langle q_1, \mathbf{s}(a, p_1, \ldots, p_f), q'_1 \rangle \in R_i$, $r_2 = \langle q_2, \mathbf{r}(a, \alpha_1, \ldots, \alpha_f), q'_2 \rangle \in R_j$ s.t. q_1 is the state of a process of type A, q_2 is the state of a process of type B, $A \not\rightharpoonup B$, we define a rule:

$$ok, q_1(x_1, \ldots, x_r), q_2(y_1, \ldots, y_r) \to ok, err(x'_1, \ldots, x'_r), q'_2(y'_1, \ldots, y'_r) : \varphi$$

where φ is the constraint $\varphi_1, \ldots, \varphi_f, \psi$ defined as follows: for $i : 1, \ldots, f$, if $\alpha_i = ?p_j$, then φ_i is the equality $y_i = x_j$ (each guard must be satisfied), if $\alpha_i = \downarrow p_j$ or $\alpha_i = *$, then $\varphi_i = true$; Furthermore, $\psi = \wedge_{i=1}^{r} x_i' = x_i, y_i' = y_i$ to denote that values of registers remain unchanged.

Finally, for each pair $C \not\downarrow D$ we define a rule $h_C(x), h_D(x) \rightarrow conflict$ to detect conflicts over data that have been stored in incompatible processes. We show an example of encoding (and analysis) of the model in Fig. 1 in Section 5 in appendix.

By construction of our reduction, we obtain that $\gamma_0[h_0]$ reaches configuration $\gamma[h]$ iff $init \triangleright N_\gamma \oplus h$, where N_γ is the multiset that contains ok, and, for each node u of type A in γ, a formula $q(\overline{v})$ where $q = L_Q(\gamma, u)$, and $\overline{v} = L_M(\gamma, u)$.

In the above construction we use the monadic predicate $h_C(x)$ to maintain footprints of identifier x received by an instance of a process of type C. The last rule is used to detect conflicts between two footprints for the same identifier. Footprints work as records of an infinite memory that associates to every identifier all processes visited during an execution. We need here infinite memory since our decision problems consider any possible initial configuration.

▶ **Theorem 7.** *The $CD(1,1)$ problem is decidable.*

Proof. For $r = 1$, the rewriting rules resulting from the encoding of the $CD(1,1)$ problem into MSR(Id) consist of monadic predicates only. By construction, $CD(1,1)$ holds if and only if from the MSR(Id) configuration $init$ it is possible to reach a configuration N s.t. $conflict$ belongs to N. This is an instance of the coverability problem for MSR(Id) that is decidable for monadic rewriting rules as shown in [11]. ◀

Besides decidability, the encoding can still be applied to obtain a possibly non terminating procedure for solving conflict detection for $r > 2$. The procedure is based on the symbolic backward reachability procedure for MSR(C) specifications described in [11].

▶ **Theorem 8.** *The $VD(1,1)$ problem is decidable.*

Proof. We use the same reduction to MSR(Id) used for conflict-detection and add rules of the form $err(x_1, \ldots, x_n) \rightarrow err : true$ to detect error states in individual processes. By construction, it follows that $VD(1,1)$ holds if and only if from the configuration $init$ we can reach a configuration N s.t. $err \in N$. ◀

Our Example Let us first expand the encoding of the four compontents of Fig. 1 in MSR(Id). Since each component is defined by send/receive operations only, the MSR(Id) model consists of the following rewriting rules: $c_1(x), a_1(y), ok \rightarrow c_1(x), a_2(x), h_a(x), ok : true$, $a_2(x), b_1(y), ok \rightarrow a_3(x), b_2(x), h_b(x), ok : true$, and $b_2(x), i_1(y), ok \rightarrow b_3(x), i_1(x), h_i(x), ok : true$ where c_1 is the single state of process type C, a_1, a_2, a_3 are the states of process type A, b_1, b_2, a_3 are the states of process type B, and i_1 is the single state of process type I. Initial configurations consists of nodes with distinct identifiers in their registers. They can be generated by a finite set of MSR(Id) rules described in appendix. Finally, rule $h_C(x), h_I(x) \rightarrow conflict : true$ specifies a conflict detection due to information leaking. Checking for possible detection can be done by executing a symbolic backward exploration that exploits constrained multisets like $M = h_C(x), h_I(x) : true$ as a symbolic representation of all possible configurations containing instance of M. The computation of predecessors can be done symbolically. Termination of predecessor computation is guaranteed by the well-structured property of monadic MSR(Id) proved in [11]. For the considered example, we apply a CLP-based implementation of the engine. As a proof of concept, experimental results on the considered example are given in appendix.

5 Proof of Concept

Let us go back to the Example of Fig. 1. In order to detect violations without run-time permission violations, we have to verify conflict detection for arbitrary initial configurations. In our abstraction of activities, we just need one register for component used to store received data. The content component contains an identifier associated to the device private data. The problem can thus be decide via: (1) an encoding in MSR(Id); (2) by applying the symbolic backward reachability algorithm defined in [10]. Let us first expand the encoding of the four compontents of Fig. 1 in MSR(Id). Since each component is defined by send/receive operations only, the MSR(Id) model consists of the following rewriting rules:

$$c_1(x), a_1(y), ok \rightarrow c_1(x), a_2(x), h_a(x), ok : true$$
$$a_2(x), b_1(y), ok \rightarrow a_3(x), b_2(x), h_b(x), ok : true$$
$$b_2(x), i_1(y), ok \rightarrow b_3(x), i_1(x), h_i(x), ok : true$$

where c_1 is the single state of process type C, a_1, a_2, a_3 are the states of process type A, b_1, b_2, a_3 are the states of process type B, and i_1 is the single state of process type I.

Since we are interested in conflict detection, we omit here the rules that encode permission violations (i.e. those leading to *error* states). The initial configuration is defined via the following rules:

$$init \rightarrow init, max(x) : true$$
$$init, max(x) \rightarrow c_1(x), max(y) : x < y$$
$$init, max(x) \rightarrow a_1(x), max(y) : x < y$$
$$init, max(x) \rightarrow b_1(x), max(y) : x < y$$
$$init, max(x) \rightarrow i_1(x), max(y) : x < y$$
$$init, max(x) \rightarrow ok : true$$

Finally, the following rule specifies a conflict detection due to information leaking from the content- to internet-component.

$$h_c(x), h_i(x) \rightarrow conflict$$

Checking for possible detection can be done by executing a symbolic backward exploration that exploits the constrained multiset $h_c(x), h_i(x) : true$ as a symbolic representation of all possible larger configurations containing instances of C. The computation of predecessors is fully automated. Furthermore, termination is guaranteed by the well-structured property of monadic MSR(Id) proved in [11].

For the considered example, we perform the following experiments. First of all, the rewriting rules are represented in Prolog as the following set of facts.

```
rule([c1(X),a1(_)],[c1(X),a2(X),ha(X)],{},1).
rule([b1(_),a2(X)],[b2(X),a3(X),hb(X)],{},2).
rule([b2(X),i1(_)],[b3(X),i1(X),hi(X)],{},3).
```

We omit here the initialization phase to simplify the analysis (e.g. we can omit the `ok` predicate). The seed of backward search is the fact `f(0, [hc(A), hi(A)], {}, 1, 0, 0)`. A fact $f(i, m, c, n, r, f)$ denotes a multiset constraint $m : c$ computed at step i of the analysis, with order number n, obtained by applying rule r backwards to a non-deterministically chosen submultiset of the multiset constraint contained in fact f. Each fact $f(i, m, c, v_1, v_2, v_3)$ is a representation of an infinite set of configurations obtained by first taking an instantiation m_1 of the formula $m : c$ and then by taking any multiset $m' = m_1 \oplus m_2$ for any multiset m_2.

The symbolic backward engine computes all predecessors in three steps:

```
f(3, [c1(A),a1(_),b1(_),i1(_),hc(A)], {}, 4, 3, 1).
f(2, [b1(_),a2(A),i1(_),hc(A)], {}, 3, 2, 2).
f(1, [b2(A),i1(_),hc(A)], {}, 2, 1, 3).
f(0, [hc(A),hi(A)], {}, 1, 0, 0).
```

The constraint `{}` is equivalent to *true*. The symbol `\_` corresponds to an anonymous free variable. Initial configurations are contained in the resulting infinite set of configurations. From the fixpoint, we can build a trace from an initial configuration to a conflict. We just have to follow the history of the predecessor computation. Fact 4 is generated from fact 3 via rule 1. Fact 3 is generated from fact 2 via rule 2. Fact 2 is generated from fact 1 via rule 3. In the trace we can verify that an identifier can move from an instance of a content component to an instance of an internet component yielding a violation that cannot be detected by using the underlying permission model.

To avoid conflicts, we can modify the definition of the A and B processes so that the start method is invoked without adding data in the intent. The resulting rules (in Prolog notations) are as follows.

```
rule([c1(X),a1(_),,ok],[c1(X),a2(X),hc(X),ha(X),ok],{},1).
rule([b1(Z),a2(X),ok],[b2(Z),a3(X),ok],{},2).
rule([b2(X),i1(_),hp(X),ok],[p3(X),i1(X),hp(X),hi(X),ok],{},3).
```

In the second rule instances of A and B synchronize with no data exchange (each process keeps the old value in its register). Via the analysis with backward search, we now get the following fixpoint:

```
f(3, [c1(_),a1(_),ok,b1(A),i1(_),hc(A)], {}, 4, 3, 1).
f(2, [b1(A),a2(_),ok,i1(_),hc(A)], {}, 3, 2, 2).
f(1, [b2(A),i1(_),ok,hc(A)], {}, 2, 1, 3).
f(0, [hc(A),hi(A)], {}, 1, 0, 0).
```

Fact 3 has only instances in initial states ($c1$, $a1$, $p1$, $i1$) thus is candidate to contain denotations of initial configurations. However in fact 3, $b1$ of type B has an identifier shared with footprint hc associated to type C. By definition, in initial configurations each identifier has the type associated to process in which it is stored. Thus, no instance of the pattern represented by fact 3 can be an initial state. Namely, any multiset $m \oplus m'$ s.t. m is an instances of `[c1(_),a1(_),ok,p1(A),i1(_),hc(A)]` cannot be an initial state. The same holds for fact 0, its denotation cannot contain initial configurations (it is not possible that the same identifier belongs to different footprints in an initial configuration). Since symbolic backward reachability generates all symbolic predecessors of upward closed sets of configurations, the fixpoint is a proof that the modified model is conflict-free for any number of nodes in initial configurations.

The encoding can be extended to process with multiple data fields. As an example, consider processes with two registers: identifier and data. Processes have now the following rewriting rules:

$$c_1(id, x), a_1(id1, y), ok \rightarrow c_1(id, x), a_2(id1, x), h_a(x), ok : true$$
$$a_2(id, x), b_1(id1, y), ok \rightarrow a_3(id, x), b_2(id1, x), h_b(x), ok : true$$
$$b_2(id, x), i_1(id1, y), ok \rightarrow b_3(id, x), i_1(id1, x), h_i(x), ok : true$$

We remark that the footprints are still defined by monadic predicates. Backward search can still be applied.

6 Conclusions and Related Work

We have presented a framework for reasoning about abstract models of concurrent systems in which interaction is regulated by a statically defined permission model. In this setting we have studied computational issues of two fundamental problems: detecting permission violations and conflicts due to value passing. The problems are formulated in sucha a way to capture properties for concurrent systems with an arbitrary number of components. Our model is inspired by the automata-based model of distributed systems proposed in [14, 15, 13] to study of robustness of broadcast communication in unreliable networks with different types of topology and different types of dynamic reconfigurations. Verification of broadcast protocols in fully connected networks in which nodes and messages range over a finite set of states has been considered, e.g., in [16, 18, 4, 17, 24]. Differently from the above mentioned works, the focus of the present paper is the analysis of process interaction, via rendez-vous and value passing, controlled by permission models and of its interplay with data. Parameterized verification of provenance in distributed applications has been considered in [20] where regular languages are used as a formal tool to symbolically analyze the provenance of messages taken from a finite alphabet. In the present paper we consider messages from an infinite alphabet and use transition systems with histories that share information in common with the current state. The use of predicates to observe the history of data shares similarities with approaches based on the use of policy automata and type systems. Type systems have been defined in abstract languages that model Android applications in [2, 9] based on the history expressions introduced in [5, 6]. Policy automata are automata with management of names that can be used to specify policies for accessing resources. They can be analyzed by using model checking algorithms for BPAs [7]. Concerning validation of updates of access control policies, parameterized reasoning via constraint and SMT solvers has been considered, e.g., in [25, 3, 21, 22]. Register Automata and History-Register Automata have also been used to model programs with dynamic allocation in [26, 27]. To our knowledge, the use of history predicates that share information with the current state and the application of well-structured transition systems to verify data tracking in parameterized concurrent systems are two novel ideas. From a technical point of view, our results are obtained via reductions to low level concurrency models like Petri nets and rewriting systems in which it is possible to manipulate data taken from an infinite ordered domain of identifiers like MSR(Id) [8, 11, 1]. MSR(Id) is also strictly related to ν-nets [23] that provide fresh name generation and equality constraints. The relation between MSR(Id) and ν-nets is studied in [12]. As shown in [1], the MSR(Id) model is strictly more expressive than Petri Nets and it has the same expressive power of Datanets [19], an extension of Petri Nets with ordered data.

References

1 P. A. Abdulla, G. Delzanno, and L. Van Begin. A classification of the expressive power of well-structured transition systems. *Inf. Comput.*, 209(3):248–279, 2011.

2 A. Armando, G. Costa, and A. Merlo. Formal modeling and reasoning about the android security framework. In *TGC*, pages 64–81, 2012.

3 A. Armando and S. Ranise. Scalable automated symbolic analysis of administrative role-based access control policies by SMT solving. *J. of Computer Security*, 20(4):309–352, 2012.

4 T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV*, pages 221–234, 2001.

5 M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Types and effects for resource usage analysis. In *FOSSACS*, pages 32–47, 2007.
6 M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Local policies for resource usage analysis. *ACM TOPLAS*, 31(6), 2009.
7 M. Bartoletti, P. Degano, G.L. Ferrari, and R. Zunino. Model checking usage policies. *MSCS*, 25(3):710–763, 2015.
8 M. Bozzano. A Logic-Based Approach to Model Checking of Parameterized and Infinite-State Systems, PhD Thesis, DISI, University of Genova, June 2002.
9 M. Bugliesi, S. Calzavara, and A. Spanò. Lintent: Towards security type-checking of android applications. In *FMOODS/FORTE*, pages 289–304, 2013.
10 G. Delzanno. An overview of MSR(C): A clp-based framework for the symbolic verification of parameterized concurrent systems. *ENTCS*, 76:65–82, 2002.
11 G. Delzanno. Constraint-based automatic verification of abstract models of multithreaded programs. *TPLP*, 7(1-2):67–91, 2007.
12 G. Delzanno and F. Rosa-Velardo. On the coverability and reachability languages of monotonic extensions of petri nets. *Theor. Comput. Sci.*, 467:12–29, 2013.
13 G. Delzanno, A. Sangnier, R. Traverso, and G. Zavattaro. On the complexity of parameterized reachability in reconfigurable broadcast networks. In *FSTTCS'12*, pages 289–300, 2012.
14 G. Delzanno, A. Sangnier, and G. Zavattaro. Parameterized verification of ad hoc networks. In *CONCUR'10*, pages 313–327, 2010.
15 G. Delzanno, A. Sangnier, and G. Zavattaro. On the power of cliques in the parameterized verification of ad hoc networks. In *FOSSACS*, pages 441–455, 2011.
16 E. A. Emerson and K. S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *LICS*, pages 70–80, 1998.
17 J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *LICS'99*, pages 352–359, 1999.
18 S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.
19 R. Lazic, T. Newcomb, J. Ouaknine, A. W. Roscoe, and J. Worrell. Nets with tokens which carry data. *Fundam. Inform.*, 88(3):251–274, 2008.
20 R. Majumdar, R. Meyer, and Z. Wang. Provenance verification. In *RP*, pages 21–22, 2013.
21 S. Ranise. Symbolic backward reachability with effectively propositional logic - applications to security policy analysis. *FMSD*, 42(1):24–45, 2013.
22 S. Ranise and R. Traverso. ALPS: an action language for policy specification and automated safety analysis. In *Security and Trust Management*, pages 146–161, 2014.
23 F. Rosa-Velardo and D. de Frutos-Escrig. Decidability results for restricted models of petri nets with name creation and replication. In *Petri Nets*, pages 63–82, 2009.
24 P. Schnoebelen. Revisiting ackermann-hardness for lossy counter machines and reset petri nets. In *MFCS'10*, volume 6281 of *LNCS*, pages 616–628. Springer, 2010.
25 S. D. Stoller, P. Yang, M. I. Gofman, and C. R. Ramakrishnan. Symbolic reachability analysis for parameterized administrative role-based access control. *Computers & Security*, 30(2-3):148–164, 2011.
26 N. Tzevelekos. Fresh-register automata. In *POPL 2011*, pages 295–306, 2011.
27 N. Tzevelekos and R. Grigore. History-register automata. In *FOSSACS 2013*, pages 17–33, 2013.