Model Checking Dynamic Distributed Systems

C. Aiswarya
aiswarya.cyriac@it.uu.se

Uppsala University, Sweden

Abstract. We consider distributed systems with dynamic process creation. We use data words to model behaviors of such systems. Data words are words where positions also contain some data values from an infinite domain. The data values are seen as the process identities. We use an automata with a stack and registers to model a distributed system with dynamic process creation. The non-emptiness checking of these automata is NP-Complete. While satisfiability of first order logic over data words is undecidable, we show that model checking such an automata against full MSO logic (with data equality and comparison predicates) is decidable.

1 Introduction

Distributed systems with a pre-defined finite set of processes have been studied extensively. However, verification of distributed systems with unbounded set of processes or those with dynamic process creation has received relatively little attention. One reason might be the additional difficulty in modeling and model checking caused by the unbounded set of processes. Most of the distributed systems we encounter in our everyday life, like internet, creates processes dynamically. Hence verification of distributed systems with dynamic process creation has become a necessity, needless to say it is interesting in its own with the scope of extending the frontiers from bounded number of processes to a dynamic setting.

There has been an increasing interest in the verification of systems with unbounded number of processes in the recent years. Most of these works describe the system by describing the local processes in the system. For obtaining decidability in the shared memory setting 1) the processes are assumed to be anonymous, and 2) often a bound on the number of context switched per process is assumed to obtain decidability (cf. [3]). In the message passing setting, such systems have been considered as parametrised systems (see [1, 6, 9, 8]). In the parametrized setting, each individual process has a finite control and fixed set of registers to store the identities of some of the other processes and possibly a stack to model recursion.

A global description of such systems is interesting, mainly for protocol specifications. In [14], grammars were used to model global descriptions of systems with dynamic process creation. It was shown that model checking these grammars against MSO is decidable. In [10] the authors study how to synthesize the local implementations of processes from a global grammar specification.

Instead of the grammars of [14], a very powerful automata based formalism was proposed in [7] for global descriptions of systems with dynamic process creation. The formalism called data-multi-pushdown automata has registers and *multiple* stacks which can store (an unbounded number of) process identities. This model supported dynamic process creation by injecting "fresh" process identities into the system (values that have not been used in the history). The main (and surprising) result of [7] states that model checking of these automata against monadic second order logic (MSO) augmented with predicates for data-(dis)equality tests is decidable.

This paper is closely related to [7]. We consider a model which is a restriction and at the same time an extension of the data-multi-pushdown system. We restrict to models with only one stack (instead of multiple stacks). On the other hand, we extend the transition guards to perform data inequality tests (in addition to the (dis)equality tests if [7]). We demonstrate the modelling power of this formalism with several examples. We show that the decidability of model checking against MSO holds for this model as well, even though the MSO has predicates for data equality/disequality/inequality tests. We study control state reachability problem for this model (as opposed to full MSO model checking) in hope of obtaining better complexity. We show that it is in fact NP complete.

2 Data words to model protocols

Notation The set of natural numbers $\{1, 2, \ldots\}$ is denoted by $\mathbb{N}$. A ranked alphabet is a pair $(A, \mathsf{arityOf})$ where A is a set, and $\mathsf{arityOf} : A \mapsto \mathbb{N}$ is a mapping. Abusing notations, we sometimes write A to denote the ranked alphabet $(A, \mathsf{arityOf})$. Given a ranked alphabet $(A, \mathsf{arityOf})$ and a (potentially infinite) set B, we denote by A_B the set $\{a(b_1, \ldots, b_n) \mid a \in A,\ n = \mathsf{arityOf}(a) \text{ and } b_i \in B\}$.

2.1 Data words

A (multi-dimensional) data word, over a finite ranked alphabet $(A, \mathsf{arityOf})$ and an infinite data domain D, is a sequence of elements from Σ_D. Consider a data word $w \in \Sigma_D^*$. By $\mathsf{symAt}(i)(w)$ we denote the letter at position i of w. For $k \leq \mathsf{arityOf}(\mathsf{symAt}(i))$, we denote the kth data value at position i by $\mathsf{dataAt}(k)(i)(w)$. For example, suppose $w = a_1(d_1^1, \ldots, d_{n_1}^1)a_2(d_1^2, \ldots, d_{n_2}^2)a_3(d_1^3, \ldots, d_{n_3}^3)\ldots$. Then $\mathsf{symAt}(i)(w) = a_i$ and $\mathsf{dataAt}(k)(i)(w) = d_k^i$ if $k < n_i$.

2.2 Dynamic Distributed Systems (DDS)

A Dynamic Distributed Systems (DDS) consists of a collection of processes. Each process in the system has a unique identifier (called pid). We fix the set of process identifiers of any DDS to be $\mathbb{N}$.

We consider DDS capable of performing two types of atomic events: a) create, and b) message. Each of these events have two participating processes. For create,

we have the creating process and the created process. For message, we have the sender and the receiver. The created process will be a fresh (non existing) process, while the creating process as well as the sender and the receiver are assumed to be already existing in the system.

We assume the pid of the created process to be bigger (in the natural order) than that of any existing process. This is in accordance with the pid assigning conventions in Unix. This convention facilitates determining which process is more recent by comparing their process ids.

A message can have the message contents, which we denote by a predefined message type and the set of (existing) process ids appearing in the message. The number of process ids that appear in a message is determined by the message type. Let the finite ranked alphabet $(\mathsf{Messages} = \{a, b, \ldots\}, \mathsf{arityOf})$ be the predefined message types. For each $a \in \mathsf{Messages}$ we have $\mathsf{arityOf}(a) \geq 2$, since two pids are required to specify the sender and the receiver. In fact $\mathsf{arityOf}(a) - 2$ gives the number of process ids that are transmitted from the sender to the receiver in a message of type a.

The set of events of DDS is given by finite ranked alphabet $(\textsc{Events} = \{\mathtt{create}\} \cup \mathsf{Messages}, \mathsf{arityOf})$. We have $\mathsf{arityOf}(\mathtt{create}) = 2$, and $\mathsf{arityOf}(a)$ for $a \in \mathsf{Messages}$ is inherited from the ranked alphabet $(\mathsf{Messages}, \mathsf{arityOf})$ defined before. An event is an element of $\textsc{Events}_{\mathbb{N}}$. The behavior of a DDS is a data word from $\textsc{Events}^*_{\mathbb{N}}$.

Example 1. Consider the following data word over $(\{\mathtt{create}, \mathtt{msg}\}, \mathsf{arityOf})$ where $\mathsf{arityOf}(\mathtt{create}) = \mathsf{arityOf}(\mathtt{msg}) = 2$.

$$w_1 = \quad \mathtt{create}(1,2) \quad \mathtt{create}(2,3) \quad \mathtt{msg}(3,1) \quad \mathtt{create}(3,4)$$
$$\mathtt{msg}(2,1) \quad \mathtt{create}(4,5) \quad \mathtt{msg}(5,1) \quad \mathtt{msg}(4,1).$$

This behaviour can be visualised graphically as depicted in Figure 1. The sequence of events taking place on process i is given in the left to right order on the horizontal line next to i. □

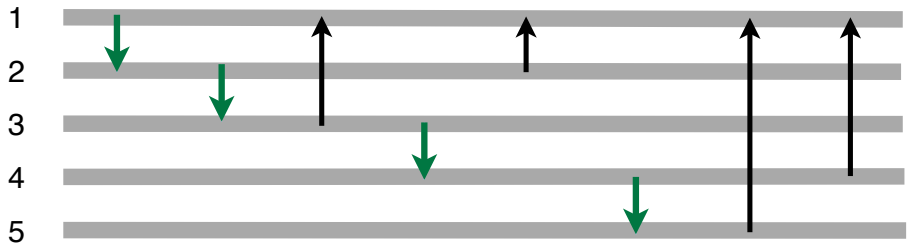

Fig. 1. A trace

Example 2 (Peer-to-peer protocol). Consider the set of data words of the form (where $m, n \in \mathbb{N}$ and $m < n$)

$$\begin{aligned} w_{n,m} = \quad & \texttt{create}(1,2) \quad \texttt{create}(2,3) \quad \ldots \quad \texttt{create}(n-1,n) \\ & \texttt{req}(n,n-1,n) \quad \texttt{req}(n-1,n-2,n) \quad \ldots \quad \texttt{req}(m-1,m,n) \\ & \texttt{msg}(m,n)\texttt{msg}(n,m) \quad \texttt{msg}(m,n)\texttt{msg}(n,m) \quad \ldots \quad \texttt{msg}(m,n)\texttt{msg}(n,m). \end{aligned}$$

These set of words describe a dynamic peer to peer protocol. The informal description of the protocol is as follows. There is a creation phase in which the processes are created in a cascade fashion. After that the last created process is in search for a peer and requests its parent to be one. It can either accept, or refuse by passing the request from its child on to its parent. This continues for a while until one process decided to be the peer, and then peer-peer communication takes place between these two (by msg events). If the request reached the process 1, it is forced to be a peer, since it cannot forward the request to its parent. Note that the messages in the request phase needs to carry the identity of the requesting process in its contents.

Figure 2 depicts the peer-to-peer protocol with $n = 6$ and $m = 2$. The data word is obtained by writing down the events in the left to right order. □

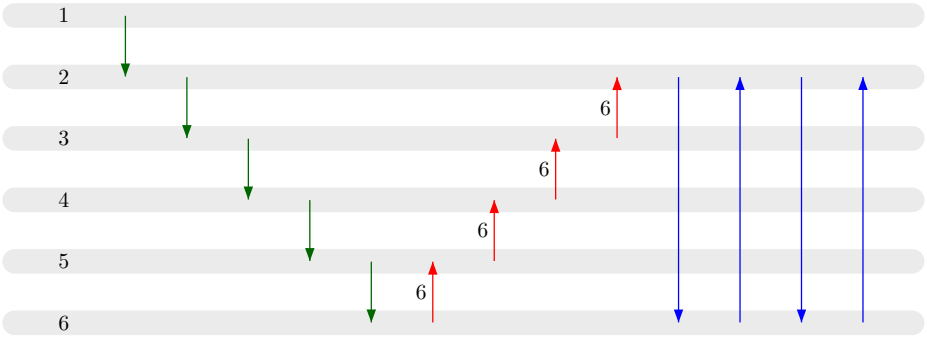

Fig. 2. A peer-to-peer protocol.

Example 3. Consider a DDS which creates processes to form a tree architecture like in Figure 3. This word can be represented by a *depth-first-search* listing of the create events. For e.g. in Figure 3 it is given by the dataword tree:

$$\begin{aligned} \mathit{tree} = \quad & \texttt{create}(1,2) \quad \texttt{create}(2,3) \quad \texttt{create}(3,4) \quad \texttt{create}(3,5) \quad \texttt{create}(2,6) \\ & \texttt{create}(1,7) \quad \texttt{create}(7,8) \quad \texttt{create}(8,9) \quad \texttt{create}(8,10) \quad \texttt{create}(10,11). \end{aligned}$$

This can be followed by a request propagating from the leftmost leaf to the right most leaf only through the leafs (i.e. the request message scans the yield of the

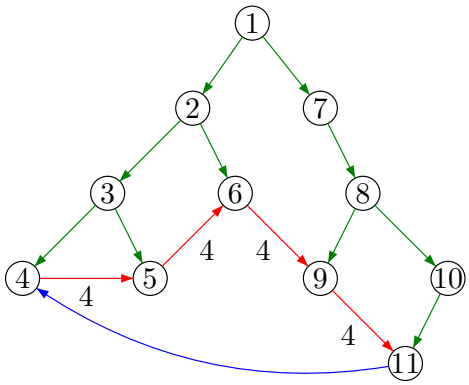

Fig. 3. A distant-relative search.

tree from left to right). This is similar to the seeking phase in the peer-to-peer protocol. A data word for this phase in the example is seek :

$$\mathit{seek} \quad = \quad \texttt{req}(4,5,4) \quad \texttt{req}(5,6,4) \quad \texttt{req}(6,9,4) \quad \texttt{req}(9,11,4).$$

Finally, the rightmost leaf (peer) sends a message directly to the leftmost leaf ($\texttt{msg}(11,4)$). Thus a data word representation of Figure 3 is $\mathit{tree}\ \mathit{seek}\ \texttt{msg}(11,4)$.

This example can be seen as modeling the search for a distant relative in a social network. The green part of the tree shows the family tree. The leaves are the current generation. The leaves know only their closest relatives in the current generation (their left and right neighbors in the left-to-right ordering of the leaves). A person in the present generation (process 4) wants to find a kin peer. The request for such a peer must be propagated along the current generation. (Older generations are perhaps dead!) □

Remark 1. The behaviors of DDS are data words over $\textsc{Events}_{\mathbb{N}}$. However any data word over $\textsc{Events}_{\mathbb{N}}$ need not have an interpretation as the behavior of a DDS. For example, $\texttt{create}(1,2)\texttt{create}(2,1)$ is a valid data word, but it cannot be seen as the behavior of a DDS since an existing process cannot be created. The dataword $\texttt{create}(2,1)$ is also not a valid behavior, since the pid of the newly created process needs to be bigger than the existing processes.

Remark 2. It might be a bit annoying to see that we have used sequences to represent the behaviors of a DDS. This looks like it captures only *linearzations* of the distributed behavior. However, as we will shortly see, the specification language we use is powerful enough to recover the concurrency information from a linearization.

2.3 Monadic Second Order logic over data words

Now we describe a powerful specification language to reason about the properties of data words. We use an extension of MSO over words to data words. In addition to the MSO over words, it allows comparison of data values.

We assume countably infinite supplies of first-order and second-order variables. We let $x, y, \ldots$ denote first-order variables, which vary over positions in the word, and we use $X, Y, \ldots$ to denote second-order variables, which vary over sets of positions in the word.

Definition 1 (MSO logic over data words). *The class* $\mathrm{MSO}_d(\textsc{Events})$ *of* monadic second-order (MSO) formulas *over data words is given by the following grammar, where a ranges over* Events*, and* k, ℓ *are at most the maximum rank of any letter in* Events*:*

$$\varphi ::= a(x) \mid d^{<}_{k,\ell}(x,y) \mid d^{=}_{k,\ell}(x,y) \mid x \leq y \mid x \in X \mid \neg\phi \mid \phi \vee \phi \mid \exists x\phi \mid \exists X\phi$$

If the free variable x is interpreted as position i of a data word w, then the formula $a(x)$ holds if $\mathsf{symAt}(i)(w) = a$. If the free variable x and y are interpreted as positions i and j respectively, then the formula $d^{=}_{k,\ell}(x,y)$ holds if $\mathsf{dataAt}(k)(i)(w) = \mathsf{dataAt}(k)(\ell)(w)$. Semantics of Formula $d^{<}_{k,\ell}(x,y)$ is similar but requires $\mathsf{dataAt}(k)(i)(w) < \mathsf{dataAt}(k)(\ell)(w)$ instead of $\mathsf{dataAt}(k)(i)(w) = \mathsf{dataAt}(k)(\ell)(w)$. Formula $x \leq y$, the boolean connectives, and quantifiers are self-explanatory. We may use the usual abbreviations $x < y$, $\forall x\phi$, $\phi \to \psi$ ...

If ϕ is a sentence, i.e., it does not have any free variable, then we set $L(\phi)$ to be the set of data words w such that $w \models \phi$.

Example 4. Consider the property that any process which requests for a peer eventually gets a peer. This can be said by the following formula: $\forall x\, req(x) \to \exists y(y > x \wedge msg(y) \wedge d^{=}_{3,2}(x,y))$. That is, if there is a "req" event, then there is a "msg" event in the future such that the parameter of "req" event and the receiver of the "msg" event are the same.

Example 5. Consider a property that the participants of any message are always leaves, i.e., they do not create other processes. This can be said by the formula $\forall x\, \neg create(x) \to \neg\exists y\, create(y) \wedge (d^{=}_{1,2}(y,x) \vee d^{=}_{1,1}(y,x))$

Example 6. Messages are always sent from younger processes to older processes can be said by the formula $\forall x\, msg(x) \to d^{<}_{2,1}(x,x)$

Example 7. Every created process eventually sends a message to the "root" process. This can be said by the formula $\exists x\,(\min(x) \wedge \forall y\,(\mathtt{create}(y) \to \exists y'\,(d^{=}_{2,1}(y,y') \wedge d^{=}_{1,2}(x,y'))))$. The formula holds in the data word w of Example 1 (Figure 1).

Example 8. This example demonstrates that the our logic is powerful enough to express causal dependencies, though it is evaluated on linearizations. The property that every two events are causally dependent can be said by the following formula. $\forall x \forall y(x \preceq y \vee y \preceq x)$ where $x \preceq y := (x \leq y \wedge \bigvee_{i,j\in\{1,2\}} d^{=}_{i,j}(x,y))^*$. We do not explicitly give this formula, but transitive closure is definable in MSO.

3 Data Pushdown automata

A data pushdown automata is a finite state automaton equipped with a stack and a finite set of registers. It can remember data values by either storing it in registers or by pushing it to the stack.

All registers except one are undefined in the beginning. The undefined registers hold a special value $\perp$. The defined registers hold the pid of the initial (root) process.

The stack symbols come from a ranked alphabet $\mathcal{Z}$, and the stack contains words from $\mathcal{Z}_{\mathbb{N}}^*$. Only the contents of those registers with a proper pid can be pushed onto the stack. Thus the stack does not contain $\perp$. Similarly the registers can be rewritten by only pids. Thus a register if ever gets to store a pid, it will never hold $\perp$ again.

At any state the automaton may (optionally) pop the topmost letter on the stack, while storing the associated data-values (pids) to some registers. Then it can perform an event involving the data values in the registers. Then it may (optionally) push another letter from $\mathcal{Z}_{\mathbb{N}}$ to the stack where the data-values come from the current register contents. Finally it reassigns the register values, and updates its state.

The infinite set of transition labels allow a finite abstraction by writing the register name which contains the data value rather than the actual data value. Let $\mathcal{R}$ be the finite set of register names. The set of such abstract events is $\textsc{Events}_{\mathcal{R}}$. That is, $\textsc{Events}_{\mathcal{R}} = \{a(r_1, \ldots r_n) \mid a \in \textsc{Events},\ n = \mathsf{arityOf}(a)$ and r_i is a register name from the set $\mathcal{R}\}$. The abstract pop and push actions can be described using $\mathcal{Z}_{\mathcal{R}}$. The letter $Z(r_1, \ldots, r_n) \in \mathcal{Z}_{\mathcal{R}}$ for a pop action means that, upon popping the letter $Z(d_1, \ldots, d_n) \in \mathcal{Z}_{\mathbb{N}}$ the data-values $d_1, \ldots, d_n$ are stored in registers $r_1, \ldots, r_n$ respectively. Similarly, for a push action denoted by the letter $Z(r_1, \ldots, r_n) \in \mathcal{Z}_{\mathcal{R}}$ means that that the letter $Z(d_1, \ldots, d_n) \in \mathcal{Z}_{\mathbb{N}}$ is actually pushed into the stack where $d_1, \ldots, d_n$ are the data-values stored in registers $r_1, \ldots, r_n$ respectively.

A subtle point in our model is the semantics of `create` event. If it executes a `create` event, the data value in the target register is rewritten by a "fresh" value which is higher than any of the data values used so for. This freshness is very crucial for our decidability results.

We define these notions formally.

Definition 2 (Data pushdown automaton). *Let $\Bbbk \geq 0$. A $\Bbbk$-register data pushdown automaton (DPA) over* Events *is a 7-tuple $\mathcal{A} = (S, \mathcal{Z}, s_0, r_0, Z_0, F, \Delta)$ where S is a finite* set of states*, $\mathcal{Z}$ is a finite ranked alphabet of* stack symbols*, $s_0 \in S$ is the* initial state*, r_0 is the initial state, $Z_0 \in \mathcal{Z}$ is the* start symbol *with $\mathsf{arityOf}(Z_0) = 0$, and $F \subseteq S$ is the set of* final states*. Moreover, Δ is a set of* transitions *of the form $\tau = (s, A(r_1, \ldots, r_n), \alpha, \mathsf{upd}, \rho, s')$ where $s, s' \in S$ are states, $A(r_1, \ldots, r_n) \in \mathcal{Z}_{\mathcal{R}}$, $\alpha \in \textsc{Events}_{\mathcal{R}}$ and $\mathsf{upd} \in \mathcal{Z}_{\mathcal{R}}^*$ and $\rho : [\Bbbk] \mapsto [\Bbbk]$ is an injective partial functions.*

We let $\mathit{Conf}_{\mathcal{A}} := S \times (\mathbb{N} \cup \{\perp\})^{\Bbbk} \times \mathbb{N} \times \mathcal{Z}_{\mathbb{N}}^*$ denote the set of *configurations* of $\mathcal{A}$. Configuration $\gamma = [s, \mathbf{r}, \mathtt{max}, w]$ with $\mathbf{r} = (d_1, \ldots, d_{\Bbbk})$ says that the current

state is s, the content of register r_i is d_i, all the data values which have already been used are at most $\mathtt{max}$, and the stack content is $w \in \mathcal{Z}_{\mathbb{N}}^*$ where we assume that the topmost symbol is written last. If some d_i is $\perp$, then the register r_i is undefined.

Now, consider a transition $\tau = (s, A(r_{i^1}, \ldots, r_{i^n}), \alpha, \mathsf{upd}, \rho, s')$. It is enabled at a configuration $\gamma = [s, \mathbf{r}, \mathtt{max}, w]$ if the conditions E1 ... E5 are satisfied.

E1 $w = w'\, A(d''_1, \ldots, d''_n)$

Before listing the remaining conditions, we first define an auxiliary register assignment $\mathbf{r}'$ which represents the effect of the pop on $\mathbf{r}$. Define $\mathbf{r}' = (d'_1, \ldots, d'_{\mathbb{k}})$ where $d'_i = \begin{cases} d''_j & \text{if } i = i^j \\ d_i & \text{otherwise.} \end{cases}$

E2 If $r_i \in \mathsf{pre\text{-}image}(\rho)$, then one of the following must hold:
- $d'_i \neq \perp$ or
- $\alpha = \mathtt{create}(\text{-}, r_i)$.

E3 If $\alpha = \mathtt{create}(r_i, -)$, then $d'_i \neq \perp$.

E4 If $\alpha = a(r_{j^1}, r_{j^2}, \ldots r_{j^\ell})$, then for all $k \in \{1, \ldots \ell\}$, we have $d'_{j^k} \neq \perp$.

E5 if $B(r_{k^1}, r_{j^2}, \ldots r_{k^m})$ is present in $\mathtt{upd}$ then for each $i \in \{k^1, \ldots, k^m\}$, either $d'_i \neq \perp$ or $\alpha = \mathtt{create}(\text{-}, r_i)$.

That is, for τ to be enabled at γ 1) the top stack symbol of γ should match that of the transition, 2) the register assignment should not overwrite a defined register with $\perp$, 3) and 4) the pids executing the event and message contents must exist (or the corresponding register names must be defined), and 4) the symbol $\perp$ is never written to the stack.

Now, we define the effect of an enabled transition τ at a configuration γ. Consider a register assignment function $\sigma : \mathcal{R} \mapsto \mathbb{N}$. We say that σ is suitable for γ and τ, if it can represent the effect of a $\mathtt{create}$ event (if applicable). That is σ is suitable for γ and τ if $\sigma(r_i) = \begin{cases} d > \mathtt{max} & \text{if } \alpha = \mathtt{create}(\text{-}, r_i) \\ d'_i & \text{otherwise.} \end{cases}$. That is, the in the case of a $\mathtt{create}$ event, the target register should be assigned a value larger than $\mathtt{max}$.

For every γ and τ there exists infinitely many suitable register assignment functions. If $\alpha = a(r_1, \ldots, r_n) \in \textsc{Events}_{\mathcal{R}}$, we let $\sigma(\alpha)$ be $a(\sigma(r_1), \ldots, \sigma(r_n))$. We lift this notion to words in $\textsc{Events}^*_{\mathcal{R}}$ as well: $\sigma(uv) = \sigma(u)\sigma(v)$.

If τ is enabled at γ and if σ is a suitable register assignment function, the automaton $\mathcal{A}$ can execute τ under σ generating $\sigma(\alpha)$. Then it moves into a new configuration $\gamma' = [s', \mathbf{r}', \mathtt{max}', w'']$ with $\mathtt{max}' = \max(\mathtt{max}, \max_i \sigma(r_i))$, $w'' = w'\sigma(\mathsf{upd})$ and $\mathbf{r}' = (\sigma(\rho^{-1}(r_1)), \ldots, \sigma(\rho^{-1}(r_{\mathbb{k}})))$ where we set $\rho(r_i) = r_i$ if $r_i \notin \mathsf{image}(\rho)$. In this case we write $\gamma \overset{\sigma(\alpha)}{\Longrightarrow}_{\sigma,\tau} \gamma'$.

A configuration of the form $[s_0, (d, \perp, \ldots, \perp), d, Z]$ with $d \in \mathbb{N}$ is called *initial*, and a configuration $[s, \mathbf{r}, d, w]$ such that $s \in F$ is called *final*. A *run* of $\mathcal{A}$ on

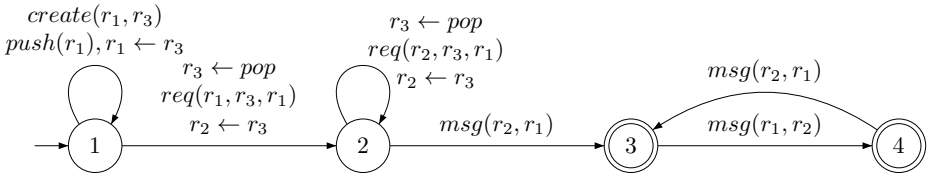

Fig. 4. A DPA for peer-to-peer

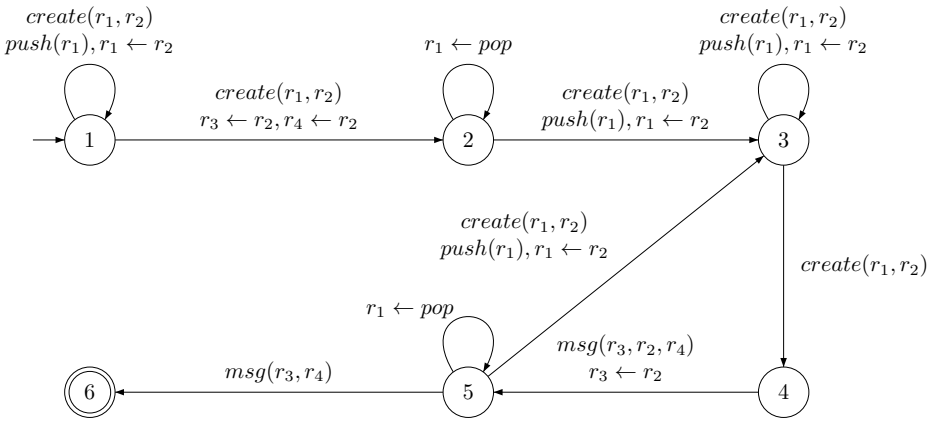

Fig. 5. A DPA for distant-relative search

$u \in \textsc{Events}^*_{\mathbb{N}}$ is a sequence $\gamma_0 \overset{\alpha_1}{\Longrightarrow}_{\sigma_1,\tau_1} \gamma_1 \overset{\alpha_2}{\Longrightarrow}_{\sigma_2,\tau_2} \cdots \overset{\alpha_n}{\Longrightarrow}_{\sigma_n,\tau_n} \gamma_n$ such that $u = \alpha_1 \cdots \alpha_n$ and γ_0 is initial. The run is *accepting* if γ_n is final. We let $L(\mathcal{A}) := \{u \in \textsc{Events}^*_{\mathbb{N}} \mid \text{there is an accepting run of } \mathcal{A} \text{ on } u\}$ be the *language* of $\mathcal{A}$.

Example 9. A DPA for the peer-to-peer protocol (cf. Example 2) is given in Figure 4. It uses three registers. We need only one extra stack symbol with $\mathsf{arityOf}\ 1$. Hence we remove this symbol in the figure for readability.

Example 10. The DPA given in Figure 5 accepts the distant-relative search example (cf. Example 3).

The *control state reachability problem* asks, given a data pushdown automata $\mathcal{A} = (S, \mathcal{Z}, s_0, r_0, Z_0, F, \Delta)$ and a state $\texttt{target} \in S$, whether there is an initial run of $\mathcal{A}$ of the form $\gamma_0 \overset{\alpha_1}{\Longrightarrow}_{\sigma_1,\tau_1} \gamma_1 \overset{\alpha_2}{\Longrightarrow}_{\sigma_2,\tau_2} \cdots \overset{\alpha_n}{\Longrightarrow}_{\sigma_n,\tau_n} \gamma_n$ where γ_n is of the form $[\texttt{target}, \mathbf{r}, d, w]$. The non-emptiness problem asks, given a data pushdown automata $\mathcal{A} = (S, \mathcal{Z}, s_0, r_0, Z_0, F, \Delta)$, whether the language of $\mathcal{A}$ is non-empty (i.e., $L(\mathcal{A}) \neq \emptyset$). The control state reachability problem and the non-emptiness problem are inter-reducible.

4 Non-emptiness of DPA

We show the complexity of non-emptiness of DPAs in this section.

Theorem 1. *Non-emptiness checking of data pushdown automata is NP-Complete*

Proof. We show the NP hardness by reducing 3-CNF-SAT to the non-emptiness problem of DPA. The problem 3-CNF-SAT which is given below is a well known NP-Complete problem. Let $V = \{v_1, \ldots v_n\}$ be a set of propositional variables. By $\overline{V}$, we denote the set $\{\overline{v_1}, \ldots, \overline{v_n}\}$, the set of negations of propositional variables. Let $\mathsf{Lit} = V \cup \overline{V}$ be the set of literals.

> Input: $\varphi \equiv \bigwedge_{i=1}^{m} C_i$ where $C_i = \ell_1^i \vee \ell_2^i \vee \ell_3^i$ and $\ell_j^i \in \mathsf{Lit}$ for $1 \leq j \leq 3$.
> Question: Is there a satisfying truth assignment of the variables V such that φ evaluates to `true`?

Our reduction is as follows. On the input φ, we construct an DPA $\mathcal{A}_\varphi$ as given in the Figure 6. Remember that ℓ_j^i is actually some v_k or $\overline{v_k}$. Thus $\mathcal{A}_\varphi$ uses $2n + 1$ registers: $\mathcal{R} = \{x_0\} \cup \mathsf{Lit}$. On going from state s_{i-1} to state s_i the run defines one and only one of the two registers v_i and $\overline{v_i}$. Thus on reaching state s_n, the configuration corresponds to a unique truth assignment: The register v_i is defined if and only if the propositional variable v_i is set to `true` by the truth assignment, and the register $\overline{v_i}$ is defined if and only if the propositional variable v_i is set to `false` by the truth assignment. Thus there is a run from s_0 to s_n corresponding to every truth assignment, and there is a truth assignment corresponding to every run from s_0 to s_n. The run can be extended to reach the state s_1' if and only if the current truth assignment satisfies the clause C_1. Inductively, the run can be extended to reach the state s_i' if and only if all the clauses $C_1, \ldots C_i$ are satisfiable by the current truth assignment. Hence there is an accepting run of the DPA $\mathcal{A}_\varphi$ if and only if ϕ is satisfiable. This proves the NP-hardness. Notice that, non-emptiness is NP hard without using the stack.

We now describe the NP algorithm.

From the DPA $\mathcal{A} = (S, \mathcal{Z}, s_0, r_0, Z_0, F, \Delta)$, we obtain a classical pushdown automata (over finite alphabet) as follows. The set of states of the pushdown automata is $S \times 2^{\mathcal{R}}$. The set of stack symbols of the pushdown automata is the unranked alphabet $\mathcal{Z}$ (i.e., only the symbols of the ranked alphabet $(\mathcal{Z}, \mathsf{arityOf})$). Intuitively, a state (s, R) corresponds to a configuration of $\mathcal{A}$ where the state is s and the set of defined registers is precisely $R \subseteq \mathcal{R}$. If the DPA has a transition $\tau = (s, A(r_{i^1}, \ldots, r_{i^n}), \alpha, \mathsf{upd}, \rho, s')$, then the pushdown automata has transition of the form $((s, R), A, \alpha, \mathsf{symAt}(\mathsf{upd}), (s', R'))$ where

- $\mathsf{symAt}(\mathsf{upd})$ corresponds to the word in $\mathcal{Z}^*$ obtained by projecting to the $\mathsf{symAt}()$ of upd.
- If α is of the form $\mathtt{create}(r_i, r_j)$ then $r_i \in R \cup \{r_{i^1}, \ldots, r_{i^n}\}$, i.e. r_i must be defined. Further $R' = R \cup \{r_{i^1}, \ldots, r_{i^n}\} \cup \{r_j\}$.
- If α is of the form $\alpha(r_{j^1}, \ldots, r_{j^m})$ where $\alpha \neq \mathtt{create}$, then for each r_{j^k}, $r_{j^k} \in R \cup \{r_{i^1}, \ldots, r_{i^n}\}$, i.e. r_{i^k} must be defined. Further $R' = R \cup \{r_{i^1}, \ldots, r_{i^n}\}$.

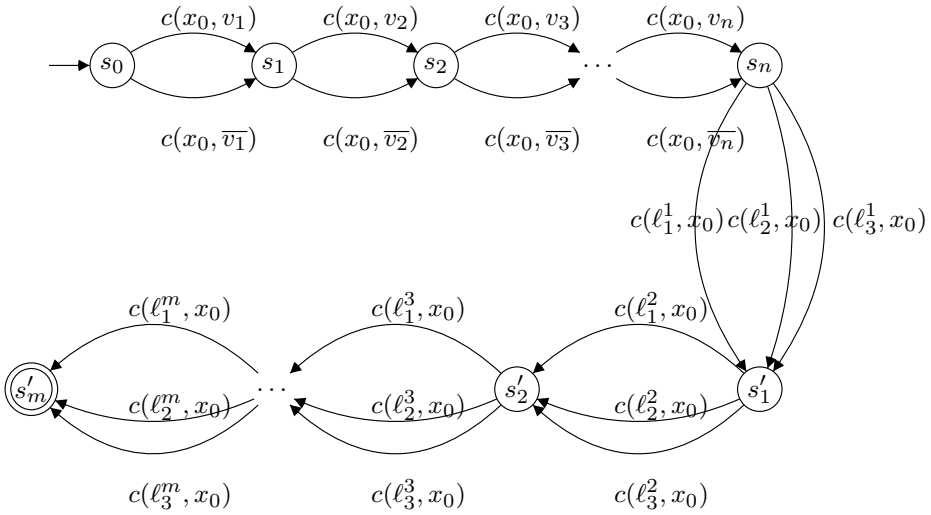

Fig. 6. Reduction from 3-CNF-SAT to non-emptiness of DPA

Since we do not have any guards with data value comparisons in the transitions of the DPA $\mathcal{A}$, if this pushdown automata has an accepting run, then the DPA also has an accepting run.

However, since the pushdown automata is exponential sized, the construction of this pushdown automata is too expensive for an NP algorithm. Hence, instead of constructing the automata, we will make some clever guesses to remain in NP.

Notice that the set of defined registers is monotonously non decreasing along any run. Our NP procedure guesses an ordering among the registers and assume that the registers are added into the "defined" set only in this order. Let $X_1 \subseteq X_2 \subseteq \ldots \subseteq X_{\mathbb{k}}$ be the sequence of defined registers in this order. That is, X_i be the set of first i registers that are defined according to this guessed order.

First we translate the given DPA into another one which on each transition either a) pops a symbol from the stack but does not push, or b) push one symbol to the stack, but does not pop, or c) does not push or pop. This translation causes only a linear blow-up in the size of the input DPA. Then, for each transition τ, we pre-compute $\min(\tau)$ the minimum set of registers needed to be defined in order to enable τ. We also compute $\mathsf{fin}(\tau)$, which is the resulting set of defined registers if τ was executed at $\min(\tau)$. The pre computation can be done in polynomial time, since we are associating just two sets to every transition of the DPA $\mathcal{A}$.

Then we have a saturation based reachability algorithm which tries to populate sets $R_i^j \subseteq S \times S$ with $1 \leq i < j \leq \mathbb{k}$. Note that, the number of sets R_i^j is $\mathbb{k}(\mathbb{k}+1)/2$. These sets can be populated simultaneously in polynomial time. The intended meaning of the set R_i^j is that, if $(s, s') \in R_i^j$, then if the all registers in X_i are defined, then the automaton can reach state s' from s, resulting in a new

defined set of registers which is exactly X_j. We explain the computation of R_i^j below.

The set R_i^j is initiated to the reflexive relation on states. Then for each pair of complementary transitions $\tau = (s, , -, A, , s')$ and $\tau' = (t, A, -, , -, t')$, the pair (s, t') is added to all sets R_i^j such that there exists i', j' with a) $\min(\tau) \subseteq X_i$, b) $X_{i'} \subseteq X_i \cup \mathsf{fin}(\tau)$, c) $(s', t) \in R_{i'}^{j'}$, d) $\min(\tau') \subseteq X_{j'}$ and e)$X_j \subseteq X_{j'} \cup \mathsf{fin}(\tau')$. After each iteration, if no new pair could be added to *any* set R_i^j, the procedure terminates as it has reached the fixed point. The number of iterations needed is polynomial ($|S|^2 \times \mathbb{k}^2$) since the maximum size of these sets is bounded. Finally, the automaton is non-empty if (s_0, s_f) is present in some R_i^j for some $s_f \in F$.

Thus our NP procedure guesses an ordering of the registers in which they are defined. Once this ordering is guessed, it verifies in polynomial time whether this guess can indeed lead to an accepting run.

Remark 3. Notice that, our convention of keeping the registers undefined in the beginning is very crucial for our NP-completeness. We could imagine a different semantics for DPA where the registers hold arbitrary but distinct values at the initial configuration. With such a definition, the non-emptiness checking odd DPA would be in P. We can indeed construct a pushdown automata abstracting away from pids. The pushdown automata checks whether a transition is enabled by checking the top symbol of the stack. The register values are irrelevant. Thus the emptiness checking of this variant of the DPA boils down to the emptiness checking of a polynomial sized pushdown automata.

5 MSO model checking

In fact, not only reachability but also model checking against powerful MSO_d turns out to be decidable for data pushdown automata.

Theorem 2. *Given a DPA $\mathcal{A}$ and an MSO_d formula ϕ, it it decidable to check whether $L(\mathcal{A}) \subseteq L(\phi)$.*

We give the proof outline in this section. The proof is essentially by abstracting the runs of a DPA as the runs of a Pushdown automata over a finite alphabet. Then the formula ϕ can be translated to an "equivalent" formula over the runs of PDA. The main challenge in obtaining a translation is to recover the data value comparisons. The proof is given in enough detail in [7]. However only data equality is considered in [7]. Hence we revisit the proof technique quickly.

We would like to see the runs of a pushdown automata as *Nested Words*. Nested words are words enriched with an additional binary relation (see Figure 7). The additional binary relation is used to match a push with the corresponding pop. We write $x \frown y$ to denote that there is a push at position x which is matched by a pop at position y. Indeed, $\frown$ is the additional binary matching relation. In order to comply with the last-in-first-out policy of stacks, we require that the nesting edges do not cross each other. For example, Figure 8 has a crossing of the additional edges, and hence it is not a nested word.

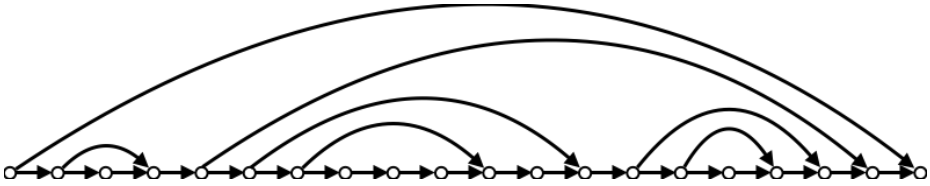

Fig. 7. A nested word

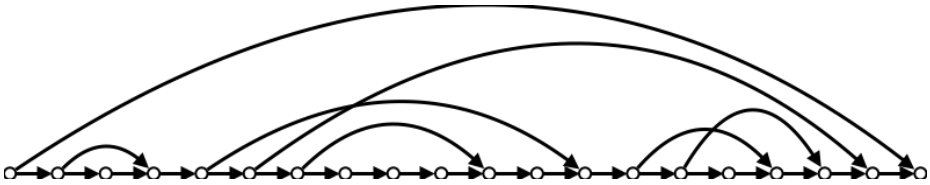

Fig. 8. This is not a nested word, since the edges do not represent last-in-first-out policy of stacks

The MSO over nested words $\mathrm{MSO}_{nw}(A)$ extends the classical MSO over words to incorporate the nesting edges. Its syntax is given by:

$$\varphi ::= a(x) \mid x \curvearrowright y \mid x \leq y \mid x \in X \mid \neg\phi \mid \phi \vee \phi \mid \exists x\phi \mid \exists X\phi$$

Here a is a letter of the finite alphabet A. We omit the obvious semantics.

MSO over nested words enjoy a decidable satisfiability problem.

Fact 3 *[2] Given a formula $\phi \in \mathrm{MSO}_{nw}(A)$, it is decidable to check whether there exists a nested word w such that $w \models \phi$.*

Any word $w \in L(\mathcal{A})$ can be embedded in an accepting run word of $\mathcal{A}$. A run word contains several consecutive nodes corresponding to a node in w, in order to carry the information of which transition it has taken. In the next step, we add the nesting edges to match a push onto the stack of the DPA with its corresponding pop. We also get rid of the real data values at this point, and keep only the register names used. This abstraction is in spirit a run of the DPA without the "register assignment" σ, enriched with the nesting edges.

The translation of the classical MSO part is via standard relativisation techniques. The data comparison is more involved. It is possible to get hold of the first position where a data value appears. We can do this by backtracking the way of this data value via registers (moving to the preceding transition via a linear edge) and stacks (moving to the transition where the value was pushed via a $\curvearrowright$ edge), and keeping the register name at which it occurred by means of second order variable. We continue the backtracking until we hit a create action with the intended register as its second argument.

Thus, we can obtain a MSO_{nw} formula $\mathsf{first}_i(x,)$ which uniquely identifies the position y at which $data_i(x)$ was created. Then $d^{=}_{k,\ell,(}(x,y)$ is equivalent to $\exists z.\mathsf{first}_k(x,z) \wedge \mathsf{first}_\ell(y,z)$. Also $d^{<}_{k,\ell,(}(x,y)$ is equivalent to $\exists z_1 z_2\, \mathsf{first}_k(x,z_1) \wedge \mathsf{first}_\ell(y,z_2) \wedge z_2 < z_1$.

Thus every MSO_d formula ϕ can be translated to an "equivalent" MSO_{nw} formula ϕ'. Indeed the set of all (abstract) valid runs of a DPA as a set of nested words is expressible in MSO_{nw}. Thus by Fact 3, Theorem 2 follows.

6 Discussions

The MSO model checking result could be extended to a DPA that runs over arbitrary data words (that is, not necessarily over `create` and `msg` alphabet). Indeed, we need to require the fresh data values to be higher than any of the previously used values. Perhaps it is also possible, instead of requiring the fresh data value to be higher, to allow guards involving data inequality comparisons of the register contents and the fresh value for the transitions.

To conclude, we have considered a special case of the Data Multi-pushdown automata defined in [7]. We have extended this restriction to include data comparison, while restricting the application domain to Dynamic Distributed Systems. This model is powerful enough to model several interesting examples. We retain all the results of [7], but also show a tight bound on the complexity of deciding non-emptiness for this particular class of automata.

References

1. P.A. Abdulla, M.F. Atig, A. Kara and O. Rezine. Verification of Dynamic Register Automata. In Raman, V and Suresh, S. P. (eds.), *FSTTCS'14*, volume 20 of *LIPIcs*, pages 653–665. Leibniz-Zentrum fr Informatik, 2014.
2. R. Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM* , 56(3), .2009.
3. M.F. Atig, A. Bouajjani, and S. Qadeer. Context-Bounded Analysis for Concurrent Programs with Dynamic Creation of Threads. In *TACAS'09*, volume 5505 of *LNCS*, pages 107–123. Springer, 2009.
4. M. Bojańczyk, C. David, A. Muscholl, Th. Schwentick, and L. Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4):27, 2011.
5. B. Bollig. An automaton over data words that captures EMSO logic. In J.-P. Katoen and B. König, editors, *CONCUR'11*, volume 6901 of *LNCS*, pages 171–186. Springer, 2011.
6. B. Bollig. Logic for Communicating Automata with Parameterized Topology. In *CSL/LICS'14*, chapter 18. ACM Press, 2014.
7. B. Bollig, A. Cyriac, P. Gastin and K. Narayan Kumar. Model Checking Languages of Data Words. In L. Birkedal (ed.), *FoSSaCS'12*, volume 7213 of *LNCS*, pages 391-405. Springer, 2012.
8. B. Bollig, P. Gastin and A. Kumar. Parameterized Communicating Automata: Complementation and Model Checking. In Raman, V and Suresh, S. P. (eds.), *FSTTCS'14*, volume 20 of *LIPIcs*, pages 625-637. Leibniz-Zentrum fr Informatik, 2014.

9. B. Bollig, P. Gastin and J. Schubert. Parameterized Verification of Communicating Automata under Context Bounds. In L *RP'14*, volume 8762 of *LNCS*, pages 45-57. Springer, 2014.
10. B. Bollig and L. Hélouët. Realizability of dynamic MSC languages. In F. M. Ablayev and E. W. Mayr, editors, *CSR'10*, volume 6072 of *LNCS*, pages 48–59, 2010.
11. S. Demri and R. Lazić. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic*, 10(3), 2009.
12. S. Demri, R. Lazić, and A. Sangnier. Model checking freeze LTL over one-counter automata. In R. M. Amadio, editor, *FoSSaCS'08*, volume 4962 of *LNCS*, pages 490–504. Springer, 2008.
13. S. Demri and A. Sangnier. When model-checking freeze LTL over counter machines becomes decidable. In C.-H. L. Ong, editor, *FoSSaCS'10*, volume 6014 of *LNCS*. Springer, 2010.
14. M. Leucker, P. Madhusudan, and S. Mukhopadhyay. Dynamic message sequence charts. In *FSTTCS'02*, volume 2556 of *LNCS*, pages 253–264. Springer, 2002.