АЛГЕБРО-ЛОГИЧЕСКИЕ МЕТОДЫ В ИНФОРМАТИКЕ
И ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

ALGEBRAIC AND LOGICAL METHODS IN COMPUTER
SCIENCE AND ARTIFICIAL INTELLIGENCE

On Termination of Transactions over Semantic Document Models [*]

A. V. Mantsivoda[1,2], D. K. Ponomaryov[2,3,4]

[1] *Irkutsk State University, Irkutsk, Russian Federation*
[2] *Sobolev Institute of Mathematics SB RAS, Novosibirsk, Russian Federation*
[3] *Ershov Institute of Informatics Systems SB RAS, Novosibirsk, Russian Federation*
[4] *Novosibirsk State University, Novosibirsk, Russian Federation*

Abstract. We consider the framework of Document Modeling, which lays the formal basis for representing the document lifecycle in Business Process Management systems. We formulate document models in the scope of the logic-based Semantic Modeling language and study the question whether transactions given by a document model terminate on any input. We show that in general this problem is undecidable and formulate sufficient conditions, which guarantee decidability and tractability of computing effects of transactions.

Keywords: Semantic Modeling, document model, transaction, chase.

1. Introduction

In [10] a Document Modeling approach has been proposed as a fundamental basis for document processing in Business Process Management

Systems (BPMS). Importantly, within this approach basic entities and primitives have been identified, which are common to BPMS such as Enterprise Resource Planning Systems, Customer Relationship Management Systems, etc. The approach rests on the natural idea that document life-cycle lies at the core of these systems. Typically, there is a static part, which describes the forms and statuses of documents (i.e., a schema), and a dynamic part, which describes changes in documents (i.e., transactions over them). In contrast to conventional architectures of BPMS, the approach of the Document Modeling shows that both parts can be given in a fully declarative fashion, thus making programming unnecessary. It suffices to describe the static part of a document model by giving a specification to document forms and fields, and to describe the dynamic part by defining transactions, their conditions, and effects. Then, given an initial state of a document model (a collection of documents), the natural problem is to compute a state (an updated collection of documents), which results from the execution of a sequence of transactions. It is argued within the Document Modeling approach that this problem can be solved with the tools of formal logic such as automated inference or model checking.

In [11], the ideas of the Document Modeling have been implemented in a logical framework in terms of the language of the Semantic Programming (also known as Semantic Modeling) [1]. It has been shown that the approach of the Document Modeling implemented this way goes beyond the common capabilities of today's Business Process Management Systems. In particular, it allows for checking document models for consistency and solving important problems like projection (e.g., what documents will be created after an accountant performs certain actions) and planning (e.g., what actions must be made in order to get an item on stock). The method follows the same line with some of the well-known approaches like Situation Calculus [13] and similar formalisms, but it addresses the topic of Business Process Management, which is a novel area of application for logic-based formalisms.

Obviously, an important question is how hard the above mentioned problems are from the computational point of view. In this respect, the key problem is computing effects of transactions over a document model. Transactions can be fired due to an input of an oracle (a user or an algorithm, which provides some input to a document model), which in turn, can cause other transactions to fire, and so on. Thus potentially, this can result in an infinite chain of updates of a document model, under which a finite resulting state is never obtained. We consider this problem in the paper and formulate a number of complexity results, which demonstrate the expressiveness of document models.

The contributions of this work are as follows. We refine the formalization of the Document Modeling given in [11] and provide a more succinct formalization in an extension of the language of the Semantic Modelling with

(non-standard) looping terms. We formulate the problem of transaction termination over document models and show that in general it is undecidable. Then we describe a sufficient condition, which guarantees decidability. For this we introduce a formal definition of a locally simple document theory (the notion previously discussed in [9]) and we show that over any such theory transaction termination is decidable. Then we estimate the complexity of computing effects of transactions and identify a case when they are polynomially bounded.

2. Preliminaries

Document Modeling follows the idea of declarative representation of documents and transactions over them. A document model consists of a description of fields, which can appear in documents (their cardinality and default values), a definition of document forms (given as collections of fields), and a definition of so called *daemons*, which specify conditions and effects of transactions and field triggers. Transactions can be fired on an input of a user or an external procedure (e.g., a Machine Learning algorithm like in [14]), or they can be fired by other transactions. Field triggers can be viewed as a special kind of transactions, but they can fire only in the event of changing a value of some document field.

The formalism of the Document Modeling includes at least three ingredients that can influence the complexity of computation. The first one is the set of operators over field values. In real-world applications of the Document Modeling, the language is restricted to basic arithmetic operations (like, summation, subtraction, etc.), which can be computed efficiently. For this reason, we do not consider the whole variety of operators over field values in the paper. We describe only basic operations and examples of their implementation in order to show that they make no contribution to the complexity of computing effects of transactions. The second ingredient is the query language used in the Document Modelling to describe collections of documents, which have certain properties. Transactions can refer to document collections given by queries and hence, the complexity of the query language influences the complexity of computing effects of transactions. We leave this effect out of the scope of this paper and focus on the complexity of transactions caused solely by their relationships to each other. For this, we adopt a simple query language implemented by predefined document filters, which can be used in the definition of transactions and are computationally simple. In the remaining part of this section, we introduce basics of the Semantic Modeling and conventions used in this paper. We refer an interested reader to [1]- [5] for details on the Semantic Modeling.

2.1. Basics of the Semantic Modeling

The language of the Semantic Modeling is a first-order language with sorts 'urelement' and 'list' , in which only bounded quantification of the following form is allowed:

−   a restriction onto the list elements $\forall x \in t$ and $\exists x \in t$;

−   a restriction onto the initial segments of lists $\forall x \sqsubseteq t$ and $\exists x \sqsubseteq t$.

where t is a list term. A list term is defined inductively via constant lists, variables of sort 'list', and list functions given below. A *constant list* (which can be nested) is built over constants of sort 'urelement' and a constant $\langle \, \rangle$ of sort 'list', which represents the empty list. The *list functions* are:

head – the last element of a non-empty list and $\langle \, \rangle$, otherwise;

tail – the list without the last element, for a non-empty list, and $\langle \, \rangle$, otherwise;

cons – the list obtained by adding a new last element to a list;

conc – concatenation of two lists;

Terms of sort 'urelement' are standard first-order terms. The predicates $\in, \sqsubseteq$ are allowed to appear in Δ_0-formulas without any restrictions, i.e., they can be used in bounded quantifiers and atomic formulas.

Formulas in the language above are interpreted over hereditarily finite *list superstructures* $HW(\mathcal{M})$, where $\mathcal{M}$ is a structure. Urelements are interpreted as distinct elements of the domain of $\mathcal{M}$ and lists are interpreted as lists over urelements and the distinguished 'empty list' $\langle \, \rangle$. In particular, the following equations hold in every $HW(\mathcal{M})$ (the free variables below are assumed to be universally quantified):

$$\neg \exists x \; x \in \langle \, \rangle$$
$$\texttt{cons}(x, y) = \texttt{cons}(x', y') \rightarrow x = x' \wedge y = y'$$
$$\texttt{tail}(\texttt{cons}(x, y)) = x, \;\; \texttt{head}(\texttt{cons}(x, y)) = y$$
$$\texttt{tail}(\langle \, \rangle) = \langle \, \rangle, \;\; \texttt{head}(\langle \, \rangle) = \langle \, \rangle$$
$$\texttt{conc}(\langle \, \rangle, x) = \texttt{conc}(x, \langle \, \rangle) = x$$
$$\texttt{cons}(\texttt{conc}(x, y), z) = \texttt{conc}(x, \texttt{cons}(y, z))$$
$$\texttt{conc}(\texttt{conc}(x, y), z) = \texttt{conc}(x, \texttt{conc}(y, z))$$

It was shown in [12] that for any appropriate structure $\mathcal{M}$, there exists a representation of its superstructure of finite lists $HW(\mathcal{M})$, in which the value of any variable-free list term t can be computed in time polynomial in the size of t (given as as string). Throughout the text, we omit subtleties related to the representation of hereditarily finite structures and we assume that for any variable-free list term t one can compute a constant list t' in time polynomial in the size of t such that $HW(\mathcal{M}) \models t = t'$, for any structure $HW(\mathcal{M})$ under consideration. For list terms $t_1, \ldots, t_n, \; n \geqslant 1$, we will use $\langle \; t_1, \ldots, t_n \; \rangle$ as a shortcut for the term

$\mathtt{cons}(\mathtt{cons}(\mathtt{cons}(\langle\,\rangle, t_1), t_2)\ldots, t_n)\ldots)$. For a list s, the notation $|s|$ stands for the number of elements in s.

In [6–8], the basic language of the Semantic Modeling was extended with non-standard list terms, which represent conditional operators (they correspond to the common 'if-then-else' or 'switch' constructs of programming languages), bounded list search, and bounded recursion (similar to the restricted 'while' operator). We refer to the obtained language as $\mathcal{L}$. The non-standard terms in $\mathcal{L}$ are called $\mathtt{Cond}$-, $\mathtt{bSearch}$- and $\mathtt{Rec}$-terms, respectively, and are defined as follows. By default any standard term in the language of the Semantic Modeling is a $\mathcal{L}$-term and any formula of the language of the Semantic Modeling is a $\mathcal{L}$-formula.

If t and $\theta(\overline{v}, x)$ is a $\mathcal{L}$-term of sort list and $\mathcal{L}$-formula, respectively, then the expression $\mathtt{bSearch}(\theta, t)(\overline{v})$ is a $\mathtt{bSearch}$-*term*. It is equal to the last element a of $t(\overline{v})$ such that $\theta(\overline{v}, a)$ holds and it is equal to $t(\overline{v})$, otherwise (i.e., if there is no such a).

If $\theta_0, \ldots, \theta_n$ are $\mathcal{L}$-formulas and $q_1, \ldots, q_{n+1}$ are $\mathcal{L}$-terms, where $n \geqslant 0$, then the term $\mathtt{Cond}[\theta_1, q_1][\theta_2, q_2]\ldots[\theta_n, q_n][q_{n+1}](\overline{v})$ is a $\mathtt{Cond}$-*term* term with the following interpretation:

$$
t(\overline{v}) = \begin{cases}
q_1(\overline{v}) & \text{if } \theta_1(\overline{v}) \\
q_2(\overline{v}) & \text{if } \theta_2(\overline{v}) \wedge \neg\theta_1(\overline{v}) \\
\ldots \\
q_n(\overline{v}) & \text{if } \theta_n(\overline{v}) \wedge \neg\theta_1(\overline{v}) \wedge \neg\theta_2(\overline{v}) \wedge \ldots \wedge \neg\theta_{n-1}(\overline{v}) \\
q_{n+1}(\overline{v}) & \text{if } \neg\theta_1(\overline{v}) \wedge \neg\theta_2(\overline{v}) \wedge \ldots \wedge \neg\theta_n(\overline{v})
\end{cases}
$$

Finally, if $f(\overline{v}), h(\overline{v}, y, z)$ and $t(\overline{v})$ are $\mathcal{L}$-terms of sort list then the expression $\mathtt{Rec}[f, h, t](\overline{v})$ is a $\mathtt{Rec}$-*term* and its value is given by $g(\overline{v}, t)$ with the following definition:

– $g(\overline{v}, \langle\,\rangle) = f(\overline{v})$
– $g(\overline{v}, cons(\alpha, b)) = h(\overline{v}, g(\alpha), b)$, for any lists α, b such that $cons(\alpha, b) \sqsubseteq t$

In this paper, we refine the formalization of the Document Modeling from [11] in the language of the Semantic Modeling extended with the above mentioned non-standard list terms. In particular, we obtain a more succinct formalization in comparison with [11]. Further in Section 3, we will introduce document theories, which formalize the key ingredients of the Document Modeling approach, and in the next section we describe conventions used in our formalization.

2.2. Conventions in Formalization of Document Theories

We use the following notions and informal conventions:
– There are pairwise disjoint finite sets *FieldNames*, *FormNames*, *FilterNames*, and *TransNames* of constants of sort *urelement*, which

provide document field, form, and filter names, and transaction names, respectively, which can be used in the axioms of a document theory.

— Natural numbers are modelled in a straightforward way as lists consisting of n empty lists, for $n \geqslant 0$, and 0 is represented by the empty list $\langle \, \rangle$ (we also show how to model real numbers in a decimal representation with a given precision).

— An *instruction* is given as a list of the form $\langle \, formName, CreateDoc \, \rangle$ (in which case it is called *CreateDoc-instruction*) or $\langle value, fieldName, docID, SetField \, \rangle$ (a *SetField-instruction*), or $\langle \, params, docID, transName \, \rangle$ (a *transaction*), where $formName \in FormNames$, $fieldName \in FieldNames$, $transName \in TransNames$, $docID$ represents a natural number, and *value*, *params* are some lists, which specify a field value and transaction parameters, respectively.

— A *queue* is a list of instructions to be executed. A queue is updated by daemons, which implement actions on the events such as changing a field value in a given document or executing a transaction. Creating a new document triggers no events.

— A *situation* is a list of instructions, which represents the history of executed instructions. The last executed instruction appears first in a situation.

— A *field* is given as a list, with the head being an element of $FieldNames$ and the tail being a list, which represents a value for a field. Every field has a default value it gets when a new document is created.

— A *document* is a list of fields (the order of fields in the list is arbitrary).

— A (document) *model* is a list consisting of tuples $\langle \, sit, form, doc, ID \, \rangle$, where ID corresponds to a natural number, doc is a document, $form \in FormNames$, and sit a situation. A model stores a version of each document in each situation which has ever taken place. The head of this list is a tuple, in which the situation is the current one, i.e., it consists of instructions (a history) that have given the model.

Situations represent contexts, in which documents are created or modified, and this information can be used in querying a document model. We note that this feature is irrelevant for the results in this paper, but we prefer to keep situations to comply with the original formalization of document models from [11]. A *document theory* consists of axioms, which specify document fields, forms, filters (i.e., the static structure of documents and query templates), and axioms for the dynamic part. The latter is given by so called *daemons* (similar to the notion used in process programming), which specify the instructions that must be executed whenever certain event happens (i.e., whenever a value of a specific field in a document is changed or a certain transaction is fired). Although formally we distinguish between CreateDoc-, SetField-instructions and transactions, we make no terminological difference between them when talking about the transaction

termination problem. The results on computing effects of transactions refer to the instructions of the form above as well.

3.   Document Theories

We define a document theory $\mathcal{T}$ as a theory in signature Σ, where Σ consists of the list functions introduced in Section 2.1 and the predicate and function symbols introduced in the axioms below. In particular, Σ contains pairwise disjoint finite subsets of constants $FieldNames$, $FormNames$, $FilterNames$, and $TransNames$, which specify field, form, filter, and transaction names, which can be used in the axioms of $\mathcal{T}$. The set $FormNames$ is supposed to be non-empty. Besides, Σ contains distinguished constants $CreateDoc$ and $SetField$, $ExecTrans$, $\texttt{fault}$, which are used to represent instructions, and $fault$ (analogous to exception in programming languages). We formulate the axioms of $\mathcal{T}$ in the language of the Semantic Modeling with non-standard terms. Initially, this language contains only two sorts: $urelement$ and $list$. For convenience, we will assume that there is also a subsort $Real$ of the sort $list$, which corresponds to (non-negative) real numbers with a given precision (denoted further as $prec$). In the following subsection, we define the sort $Real$, together with the corresponding predicates and functions, and we show how basic arithmetic operations can be implemented via list terms. In general, there are many such implementations possible, so the next subsection is best viewed as a number of introductory examples to the language of the Semantic Modeling. The only important observation is that the proposed implementation is tractable, as stated by Lemma 2 in Section 3.2. Throughout the text we assume that all the free variables in formulas are universally quantified.

3.1. Numeric Terms and Predicates

Let us define $Nat(x) \equiv \forall t \in x \; t = \langle \; \rangle$. For a natural number $n \in \omega$, denote by $\bar{n}$ the list consisting of n empty lists. Given $prec \in \omega$, $prec \geqslant 1$, we define a subsort $Real$ of the sort $list$ as follows:

$$Real(x) \quad \equiv \quad len(x) = \overline{prec} \; \wedge \; \forall t \in x \; Nat(t) \wedge len(t) \sqsubseteq \overline{9}$$

where $len(x)$ is an abbreviation for the term $Rec[\langle \; \rangle, \texttt{cons}(g(\alpha), \langle \; \rangle), x](x)$, i.e., $len(x)$ gives the number of elements in a list x. In other words, we assume that a list of sort $Real$ corresponds to the decimal representation of a real number using $prec$-many digits, for a fixed number $prec \in \omega$.

For lists x, i, let $x.i$ be a shortcut for the term

$$\texttt{Cond}[\neg Nat(i) \vee \neg(i \sqsubseteq len(x)), \; \texttt{fault}] \, [\, \texttt{Rec}[\langle \; \rangle, b, i] \,]$$

i.e., it gives the constant list `fault` if i does not correspond to a natural number or i is greater than the number of elements in x. Otherwise it gives the i-th element of x.

For lists x, y, let $x < y$ be the conjunction of $Real(x) \wedge Real(y)$ with

$$\exists i \sqsubseteq \overline{prec}\ (\ x.i \sqsubseteq y.i \wedge x.i \neq y.i\ \wedge \forall j \sqsubseteq \overline{prec}\ (i \sqsubseteq j \rightarrow x.j = y.j)\ )$$

i.e., we assume that the first digit of a real number given by a list x is `head`(x). The corresponding predicate $x \leqslant y$ is defined similarly.

For a list t, let $min(t)$ be a notation for the term

$$\texttt{Cond}[t =$$
$$= \langle\ \rangle \vee \exists s \in t(\neg Real(s)), \texttt{fault}]\ [\ \texttt{Rec}[\ \texttt{head}(t), \texttt{Cond}[b < g(\alpha), b][g(\alpha)], t]\ ]$$

The term $max(t)$ is defined similarly.

Finally, for lists x, y, let $x + y$ be a shortcut for the term

$$\texttt{Cond}[\neg(Real(x) \wedge Real(y)) \vee \texttt{tail}(sum) = \overline{1}, \texttt{fault}]\ [\texttt{head}(sum)]$$

where
$sum \equiv \texttt{Rec}[\langle\ \langle\ \rangle, \langle\ \rangle\ \rangle, \texttt{cons}(\ \texttt{tail}(s), \texttt{cons}(\texttt{head}(g(\alpha)), \texttt{head}(s))\ ), \overline{prec}]$,
   $s \equiv sumnat(\ \texttt{conc}(\ x.\texttt{cons}(\alpha, b), \texttt{tail}(g(\alpha))\ ),\ y.\texttt{cons}(\alpha, b)\ )$,
$sumnat(x, y) \equiv$
$\equiv \texttt{Cond}[\overline{10} \leqslant \texttt{conc}(x, y), \texttt{cons}(\ \overline{1}, mod10(\texttt{conc}(x, y))\ )][\texttt{cons}(\overline{0}, \texttt{conc}(x, y))]$
   and

$mod10(x) \equiv$

   $\equiv \texttt{head}(\ \texttt{Rec}[\langle\ \langle\ \rangle, \langle\ \rangle\ \rangle, \texttt{Cond}[\texttt{tail}(g(\alpha)) = \overline{10}, \texttt{cons}(\texttt{tail}(g(\alpha)),$
   $\texttt{cons}(\texttt{head}(g(\alpha)), b))]\ [\texttt{cons}(\ \texttt{cons}(\texttt{tail}(g(\alpha)), b), \texttt{head}(g(\alpha))\ )], x]\ )$.

We note that negative reals and other arithmetic operations, e.g., subtraction, multiplication, etc., can be defined in a similar fashion.

Let $prec = k + m$, where k, m are some constants, which give the length of the integer/fractional part of real numbers, respectively. For a (non-negative) real number n, let $dec(n)$ be the decimal representation of n such that the number of digits in the integer and fractional part of $dec(n)$ is exactly k and m, respectively. This is achieved by using auxiliary zeros, e.g., for $n = 3/2$ and $k = m = 2$, we have $dec(n) = 01.50$. If $dec(n)$ exists, let $List(n)$ be the list representation of $dec(n)$, i.e., the list such that $len(List(n)) = \overline{prec}$ and for all $i \in \{1, \ldots, prec\}$ and $j \in \omega$, it holds $List(n).\overline{i} = \overline{j}$ iff j is the $(prec + 1 - i)$-th digit in $dec(n)$.

The following lemma sums up the properties of the given formalization:

Lemma 1 (Implementation of Arithmetic with Precision)**.** *Let $HW(\mathcal{M})$ be a list superstructure and $prec \in \omega$ a precision. For any (non-negative) real numbers a_i such that $dec(a_i)$ exists, for $i = 1, \ldots, n$ and $n \geqslant 3$:*

$- \ dec(a_1) \propto dec(a_2) \ iff \ HW(\mathcal{M}) \models List(a_1) \propto List(a_2), \ for \propto \in \{<,=\}$
$- \ dec(a_1) + dec(a_2) = dec(a_3) \ iff \ HW(\mathcal{M}) \models \ List(a_1) + List(a_2) =$
$List(a_3)$
$- \ dec(dec(a_1) + dec(a_2)) \ does \ not \ exist \ iff \ HW(\mathcal{M}) \models \ List(a_1) +$
$List(a_2) = \mathtt{fault}$

*For $n \geqslant 1$, the value of $min(\langle \ List(a_1), \ldots, List(a_n) \ \rangle)$ or $max(\langle \ List(a_1),$
$\ldots, List(a_n) \ \rangle)$ in $HW(\mathcal{M})$ is $List(a)$ iff a is minimal/maximal among*
$dec(a_1), \ldots, dec(a_n)$, *respectively.*

3.2. Document Terms

Let us introduce notations for terms, which are used to access documents and field values in a document model.

The following term gives the last used ID for a document in a model:

$GetLastDocID(model) \equiv$
$$\equiv max(\mathtt{cons}(\mathtt{Rec}[\langle \ \rangle, \mathtt{cons}(g(\alpha), \mathtt{head}(b)), model]),$$

i.e., it implements a search for the greatest value occurring as the head of a tuple from *model* and outputs $\overline{0}$ if there are no documents in the model.

The next term gives the last version of a document (from a model) by its ID. It implements search for the last tuple with a given ID (contained in a model) and outputs the found document. If no tuple with the given ID is present in the model, the term gives $\mathtt{fault}$.

$GetDocByID(docID, model) \equiv$
$$\equiv \mathtt{Cond}[\mathtt{doctuple} = model, \ \mathtt{fault}][\mathtt{doctuple}]$$

where $\mathtt{doctuple} = \mathtt{bSearch}[\mathtt{head}(x) = docID, \ model]$.

The next term provides a field value from the last version of a document with a given ID:

$GetFieldValue(docID, fieldName, model) \equiv$
$$\mathtt{Cond}[\mathtt{document} = \mathtt{fault}, \mathtt{fault}][\ \mathtt{tail}(\mathtt{bSearch}[\mathtt{head}(x) =$$
$$= fieldName, \mathtt{document}]) \ ]$$

where $\mathtt{document} = \mathtt{head}(\mathtt{tail}(GetDocByID(docID, model)))]$.

Finally, we define the term *FindFieldPosition*, which 'splits' a document into a partitioned one (denoted as *pdocument* below), which has the form $\langle list_1, list_2 \rangle$ such that $conc(list_1, list_2) = document$ and $head(list_1)$ is a field with the required name (if there exists one in a document). This auxiliary term is employed in the axioms of a document theory to implement change of a field value in an existing document:

$FindFieldPosition(document, fieldName) \equiv$
$$\mathtt{Cond}[\mathtt{tail}(\mathtt{pdocument}) = \langle \ \rangle, \mathtt{fault}][\mathtt{pdocument}]$$

where

$$\text{pdocument} = \text{Rec}[\ \langle\ \rangle, \text{Cond}[\ \text{head}(\text{tail}(g(\alpha))) =$$
$$= fieldName, \langle\ \text{tail}(g(\alpha)), \text{cons}(\text{head}(g(\alpha)), b)\ \rangle\ ]$$
$$[\ \langle\ \text{cons}(\text{tail}(g(\alpha)), b), \langle\ \rangle\ \rangle\ ], \ document\ ]$$

Now we define by induction the notion of *document term*, which generalizes the definitions above.

Definition 1 (Document Term). *Any standard list term (i.e., which does not contain* Cond-, *bSearch, or* Rec-*terms) is a document term. If* s, t, u, i *are document terms then* $s.i$, $s + t$, $min(s)$, $max(s)$, $GetLastDocID(s)$, $GetDocByID(s, t)$, *and* $GetFieldValue(s, t, u)$ *are document terms.*
The definition of document term is complete.

An important property is that these terms are computationally tractable as stated in the following lemma.

Lemma 2 (Tractability of Document Terms). *For any* $prec \in \omega$, *document terms* $s(\overline{u})$, $t(\overline{v})$, *and vectors of constant lists* $\overline{a}, \overline{b}$:
— *a constant list* c *such that* $HW(\mathcal{M}) \models s(\overline{a}) = c$, *for any list superstructure* $HW(\mathcal{M})$ *(which contains all the urelements from* $s, t, \overline{a}, \overline{b}$*), can be computed in time polynomial in the size of* $s(\overline{a})$ *and* $\overline{prec}$;
— *it can be decided in time polynomial in the size of* $s(\overline{a})$, $t(\overline{b})$, *and* $\overline{prec}$ *whether* $s(\overline{a}) \propto t(\overline{b})$, *for* $\propto \in \{<, =\}$, *holds in any structure as above.*

Proof Sketch. The first point of the lemma is proved by induction on the form of the term s. For a standard list term, the claim readily follows from Lemma 2 in [12]. For an arbitrary document term s the claim is shown by analyzing the syntactic form of the terms $.i$, $+$, $min()$, $max()$, $GetLastDocID()$, $GetDocByID()$, and $GetFieldValue()$. It follows from their definition that each of these terms can be computed in polynomial time in the size of their parameters and $\overline{prec}$. The second point of the lemma is shown by an analysis of the definition for $<$: it gives a polynomial time algorithm to verify whether there is a segment $i \sqsubseteq \overline{prec}$, for which the condition from the definition of $<$ is true.   □

3.3. Axioms of a Document Theory

A *document theory* has the form $\mathcal{T} = \mathcal{T}_f \cup \mathcal{T}_s \cup \mathcal{T}_d$, where the theory $\mathcal{T}_f$ gives predefined filters, which can be used to select collections of documents, $\mathcal{T}_s$ gives definitions to document fields and forms (i.e., it describes the data schema, hence, the subscript s), and $\mathcal{T}_d$ describes possible transactions and triggers, their execution rules, and instruction processing rules, which generate documents or update existing ones. Thus, $\mathcal{T}_d$ describes the dynamics of documents (hence, the subscript d).

First, let us introduce auxiliary terms, which will be used in axioms of $\mathcal{T}$. The first one gives a form name of a document

$$\text{Form}(document) \equiv \texttt{head}(\texttt{tail}(\texttt{tail}(document)))$$

while the second one gives a list, in which the order of elements is reversed:

$$\text{rev}(list) \equiv \texttt{Rec}[\langle \, \rangle, \ conc(\langle \ b \ \rangle, g(\alpha)), \ list]$$

We begin with a definition of theory $\mathcal{T}_f$. For each $name \in FilterNames$, it contains a definition of a filter term of the form below. Every filter gives a list of IDs of (the last version of) those documents from a model, which satisfy conditions specified by the filter:

$$GetDocsByFilter_{name}(fName, model, params) =$$
$$= \texttt{head}(\texttt{Rec}[\langle \, \rangle, \texttt{selection}, \text{rev}(model)])$$

where $\texttt{selection}$ is a term of the form

$$\texttt{Cond}[\texttt{head}(b) \in g(\alpha), g(\alpha)] \ [\texttt{filter}(params, b), \texttt{cons}(g(\alpha), \texttt{head}(b))] \ [g(\alpha)]$$

$\texttt{filter}(params, doc)$ is a formula, which represents conditions on the documents to be selected:

$$filter(params, doc) \equiv \text{Form}(document) = fName \wedge \varphi$$

where φ is a Boolean combination of formulas of the form $s \propto t$, where $\propto \in \{<, =\}$ and s, t are document terms over variables $params$, doc such that in every term $GetLastDocID(m)$, $GetDocByID(x, m)$, or $GetFieldValue$ (x, y, m) from s or t, we have $m = model$.

Next, we define the theory $\mathcal{T}_s$. First of all, it contains axioms that describe fields and cardinalities for their values:

$$Field(x) \equiv \bigvee_{f \in FieldNames} (\ \texttt{head}(x) = f \ \wedge \ \text{Card}(\texttt{tail}(x)) \ )$$

where $\text{Card}(y)$ is a cardinality predicate, which restricts the number of elements in a list y. We consider the following cardinalities: the list is empty; it contains zero or one element (we use notation '?' for this predicate); it contains exactly one element (notation '!'); it contains one or more elements. For example, '?' is defined as

$$?(x) \equiv \forall t \in x \ \texttt{cons}(\langle \ \rangle, t) = x$$

The other predicates are defined similarly.

Further, $\mathcal{T}_s$ introduces document forms by describing which fields (with their default values) are present in a blank document of a given form:

$$Blank(name) = document \equiv$$
$$(\bigwedge_{f \in FormNames} name \neq f \ \wedge \ document = \texttt{fault})\vee$$
$$\vee \bigvee_{f \in FormNames} (name = f \ \wedge \ \varphi_f) \quad (3.1)$$

where $\varphi_f \equiv document = \langle \ \rangle$ or φ_f has the following form, for a non-empty subset $N_f \subseteq FieldNames$ (we assume that the elements of N_f are enumerated, $N_f = \{1, \ldots, n\}$):

$$\exists x_1 \in document \ldots \exists x_n \in document \bigwedge_{i \in N_f} (\texttt{head}(x_i) =$$
$$= i \wedge \texttt{tail}(x_i) = defvalue_i \wedge Field(x_i)) \wedge \forall x \in document \, (\bigvee_{i \in N_f} x = x_i)$$

where $defvalue_i$ is a list, which respects the cardinality restriction given in the definition of the $Field(x)$ predicate for $\texttt{head}(x) = i$.

The definition of the theory $\mathcal{T}_s$ is complete.

Now we are ready to define the theory $\mathcal{T}_d$. It contains definitions of daemons and a definition of a recursive $Update$ function, which given a queue, updates a model to a new state based on the definition of daemons. First, we define the $Update$ function. For the sake of readability, we split its definition into three formulas combined with disjunction and comment on them separately.

First of all, if the queue is not empty and the first instruction in the queue is not a valid one (i.e., it is neither $CreateDoc$, $SetField$ instruction, nor a transaction name $t \in TransNames$) the whole queue is skipped and the model given by the $Update$ function is the initial model. If the queue is empty, then it is assumed that all the instructions in the queue have been processed and thus, $Update$ returns the value of $model$:

$$Update(initialmodel, model, queue) = model' \equiv$$
$$(\, \texttt{head}(\texttt{head}(queue)) \notin \langle CreateDoc, SetField, \texttt{tname}_1, \ldots, \texttt{tname}_k \rangle \wedge$$
$$queue \neq \langle \ \rangle \ \wedge model' = initialmodel \,)$$
$$\vee \ (\, queue = \langle \ \rangle \wedge model' = model \,)$$
$$\vee$$
$$(3.2)$$

where $\{\texttt{tname}_1, \ldots, \texttt{tname}_k\} = TransNames$, for $k \geqslant 0$.

Otherwise the queue contains an instruction to create a document of a specific form, change a field value in a document having a certain ID, or

launch a specific transaction. In the first case, a blank document of a given form is created (which is implemented by using existential quantification) and added to the model, the instruction is removed from the queue, and the *Update* function is evaluated recursively on the resulting input. If a blank document of a form with name $formName$ can not be created (due to $formName \notin FormNames$) then the queue is skipped and *Update* returns the initial model:

$$(\; \texttt{head}(\texttt{head}(queue)) =$$
$$= CreateDoc \; \wedge \; \exists document \;\; document = Blank(\texttt{formName}) \; \wedge$$
$$(\; (document = \texttt{fault} \wedge model' = initialmodel) \; \vee (document \neq \texttt{fault} \; \wedge$$
$$model' = Update(initialmodel, \texttt{cons}(model, \texttt{newdoc}), \texttt{tail}(queue))) \;) \; \vee$$

<div align="right">(3.3)</div>

where $\texttt{formName} = \texttt{head}(\texttt{tail}(\texttt{head}(queue)))$, $\texttt{newdoc}$ is a term of the form

$$\langle \; \texttt{newsituation}, \texttt{formName}, document, \texttt{cons}(GetLastDocID(model), \langle \; \rangle) \; \rangle$$

$\texttt{newsituation} = \texttt{cons}(\texttt{Situation}(model), \langle \; \texttt{formName}, CreateDoc \; \rangle)$, and $\texttt{Situation}(model) = \texttt{head}(\texttt{tail}(\texttt{tail}(\texttt{tail}(\texttt{head}(model)))))$.

The case of $SetField$ instruction in the queue is formulated similarly, but the formalization is technically more complex, since modifying an already existing document requires more steps than creating a fresh one:

$$(\; \texttt{head}(\texttt{head}(queue)) = SetField \; \wedge (\; (\; \texttt{pdocument} = \texttt{fault} \; \vee$$
$$\neg Field(\texttt{cons}(\texttt{newFldValue}, \texttt{fldName})) \;) \wedge model' = initialmodel) \; \vee$$
$$(\texttt{pdocument} \neq \texttt{fault} \wedge model' = Update(\; initialmodel, \texttt{cons}(model,$$
$$\langle \; \texttt{newsituation}, \texttt{form}, \texttt{updateddoc}, docID \; \rangle), \texttt{extendedQueue})))$$

<div align="right">$\vee$ (3.4)</div>

where $\texttt{form} = \text{Form}(GetDocByID(\texttt{docID}, model))$, $\texttt{pdocument}$ denotes $FindFieldPosition(\texttt{head}(\texttt{tail}(GetDocByID(\texttt{docID}, model)), model), \texttt{fldName})$ and $\texttt{updatedDoc}$ is a shortcut for

$$\texttt{conc}(\texttt{tail}(\texttt{tail}(\texttt{pdocument})), \texttt{cons}(\texttt{head}(\texttt{pdocument}),$$
$$\texttt{cons}(\texttt{newFldValue}, \texttt{fldName})))$$

in which

$\texttt{docID} = \texttt{head}(\texttt{tail}(\texttt{head}(queue)))$

$\texttt{fldName} = \texttt{head}(\texttt{tail}(\texttt{tail}(\texttt{head}(queue))))$

$\texttt{newFldValue} = \texttt{head}(\texttt{tail}(\texttt{tail}(\texttt{tail}(\texttt{head}(queue)))))$

$\texttt{newsituation} = \texttt{cons}(\texttt{Situation}(model),$
$$\langle \; \texttt{newFldValue}, \texttt{fldName}, \texttt{docID}, SetField \; \rangle)$$

$\texttt{Situation}(model) = \texttt{head}(\texttt{tail}(\texttt{tail}(\texttt{tail}(\texttt{head}(model)))))$

(recall the instruction modeling conventions).

Finally, $\texttt{extendedQueue}$ is a shortcut for

$$SetFieldTrigger(docID, fieldName, newFieldValue,$$
$$\texttt{tail}(queue), model)$$

Thus, $\texttt{updatedDoc}$ is a document with an updated field value and $\texttt{extendedQueue}$ is a sequence of instructions provided by a trigger on a field value change. By the definition above, the whole queue is skipped whenever there is no field with the specified name in a given document. Note that in this case $\texttt{tail}(\texttt{pdocument}) = \texttt{fault}$ holds by the definition of $FindFieldPosition$ term.

Finally, if $\texttt{head}(\texttt{head}(queue))$ is a transaction name, a call to the daemon is made, which defines the corresponding transaction:

$$\bigvee_{tName \in TransNames} (\ \texttt{head}(\texttt{head}(queue)) = tName \land model' =$$

$$Update(initialmodel, model,$$

$$ExecTrans(tName, \texttt{docID}, \texttt{params}, \texttt{tail}(queue), model)))\ )$$

(3.5)

where $\texttt{docID} = \texttt{head}(\texttt{tail}(\texttt{head}(queue)))$ is a document, for which the transaction is to be executed, and $\texttt{params}=\texttt{head}(\texttt{tail}(\texttt{tail}(\texttt{head}(queue))))$ specifies parameters for the transaction.

Now we are in the position to define functions, which implement daemons. Their purpose is to extend the queue with a sequence of instructions depending on whether a field value in an existing document is changed or a transaction is fired. Both functions have similar definitions:

$$SetFieldTrigger(docID, fName, fValue, queue, model) \equiv \Phi$$
$$ExecTrans(tName, docID, params, queue, model) \equiv \Psi$$

where

$$\Phi = \texttt{Cond}[\theta_1, q_1], \ldots, [\theta_n, q_n][queue]$$

and for all $i \in \{1, \ldots, n\}$, $n \geqslant 0$, θ_i is a condition of the form

$\text{Form}(GetDocByID(docID, model))=\texttt{formName} \land fName=\texttt{fieldName} \land \varphi$

(in this case θ_i is called
$(\texttt{formName}, \texttt{fieldName})\textbf{-}\textit{condition}$) such that $\texttt{formName} \in FormNames$, $\texttt{fieldName} \in FieldNames$ and $q_i=\texttt{conc}(queue, \texttt{instr}_i)$,
where φ is a Boolean combination of formulas of the form $\texttt{val}_1 \propto \texttt{val}_2$,

where $\propto \in \{<, =\}$, and $\mathtt{val}_1, \mathtt{val}_2$ are document terms over variables $docID$, $fValue$, $model$ and $\mathtt{instr}_i$ (called ***queue extension***) is a list

$$\mathtt{conc}(s_1, \mathtt{conc}(s_2 \ldots \mathtt{conc}(s_{k-1}, s_k) \ldots)$$

such that

- each s_i for $i = 1, \ldots, k$, $k \geqslant 1$ (called ***instruction term***) is a list term of the form $\langle\ \langle\ \mathtt{val}, \mathtt{fieldName}', docID, SetField\ \rangle\ \rangle$ or $\mathtt{Rec}[\langle\ \rangle, h,$ $\mathtt{DocFilter}]$ with the definition: $g(\langle\ \rangle) = \langle\ \rangle$, $g(\mathtt{cons}(\alpha, id)) = h(id)$, where $h(id) = \mathtt{conc}(\langle\ \langle\ \mathtt{params}, id, \mathtt{transName}\ \rangle\ \rangle,\ g(\alpha))$, for all α, id such that $cons(\alpha,\ id) \sqsubseteq \mathtt{DocFilter}$
- $\mathtt{params}$ is a list of the form $\langle\ t_1, \ldots, t_m\ \rangle$, for $m \geqslant 0$, where every t_i is a document term over variables $docID, fValue, model$
- $\mathtt{fieldName}' \in FieldNames$ and $\mathtt{val}$ is a document term over variables $docID, fValue, model$
  (then s_i is called $(\mathtt{formName}, \mathtt{fieldName}')$-***instruction***)
- $\mathtt{DocFilter} = GetDocsByFilter_{\mathtt{name}}(\mathtt{frmName}, model, \mathtt{p})$
- $\mathtt{p}$ is a document term over variables $docID, fValue, model$
- $\mathtt{name} \in FilterNames$, $\mathtt{frmName} \in FormNames$, and $\mathtt{transName} \in TransNames$ (then s_i is called $(\mathtt{frmName}, \mathtt{transName})$-***instruction***)

Thus, changing a field value in a document may cause addition of instructions to the queue, which change other fields in the same document or execute transactions over sets of documents defined by filters.

The formula Ψ is defined similarly, but with the following minor modification (we use the notations above):
- every condition θ_i has the form

$$\mathrm{Form}(GetDocByID(docID, model)) =$$
$$= \mathtt{formName} \wedge tName = \mathtt{transName} \wedge \varphi$$

(in this case θ_i is called $(\mathtt{formName}, \mathtt{transName})$-***condition***), where $\mathtt{transName} \in TransNames$
- every s_i is a list term of the form $\langle\langle\ \mathtt{val}, \mathtt{fieldName}', docID, SetField\ \rangle\rangle$ or $\mathtt{Rec}[\langle\ \rangle, h, \mathtt{DocFilter}]$, where h is given as $\mathtt{conc}(\langle\ \langle\ \mathtt{params}', id,$ $\mathtt{transName}\ \rangle\ \rangle,\ g(\alpha))$ or $\mathtt{conc}(\langle\ \langle\ \mathtt{frmName}, CreateDoc\ \rangle\ \rangle,\ g(\alpha))$, or s_i is of the form $\langle\ \langle\ \mathtt{frmName}, CreateDoc\ \rangle\ \rangle$
  (in the latter two cases s_i is called $(\mathtt{frmName}, CreateDoc)$-***instruction***)
- $\mathtt{val}, \mathtt{val}_1, \mathtt{val}_2, \mathtt{p}$ are document terms over variables $docID, params,$ $model$ and $\mathtt{params}'$ is a list of the form $\langle\ t_1, \ldots, t_m\ \rangle$, for $m \geqslant 0$, where every t_i is a document term over variables $params, model$.

Thus, executing a transaction over a document may cause addition of instructions to the queue, which change fields in the document, create new documents (of the same or different document form), or execute transactions over sets of documents defined by filters.

The definition of the document theory $\mathcal{T}$ is complete.

Let the *size* of $\mathcal{T}$ be the total size of its axioms (given as strings).

4. Termination of Transactions

The recursive definition of $Update$ function yields the natural notion of *chase* operator, which for a given document theory $\mathcal{T}$ and constant lists $model, queue$, where $queue \neq \langle\,\rangle$, outputs lists $model'$ and $queue'$ obtained after processing the first instruction from $queue$ (i.e., $\texttt{head}(queue)$). In other words, for any list superstructure $HW(\mathcal{M})$, it holds

$$HW(\mathcal{M}) \models Update(model, model, queue) = Update(model, model', queue')$$

where $model'$ is obtained from $model$ by the definition of $Update$ function in $\mathcal{T}$ without applying recursion and either $queue'$ is obtained in the same way from $queue$, or it holds that $queue = \langle\,\rangle$. We denote this fact as $\langle\, model, queue\,\rangle \mapsto \langle\, model', queue'\,\rangle$. A *chase sequence* wrt $\mathcal{T}$ for a list $\langle\, m_0, q_0\,\rangle$ of the form above is a sequence of lists $\langle\, m_0, q_0\,\rangle, \langle\, m_1, q_1\,\rangle, \ldots$, where $\langle\, m_i, q_i\,\rangle \mapsto \langle\, m_{i+1}, q_{i+1}\,\rangle$, for all $i \geqslant 0$. A chase sequence is *terminating* if it is of the form $s_0, \ldots, s_k$, for some $k \geqslant 1$, where $s_k = \langle\, m_k, \langle\,\rangle\,\rangle$.

In the following, we note that there may not exist a terminating chase sequence for a given list $\langle\, m, q\,\rangle$ and a theory $\mathcal{T}$. Then we formulate a sufficient condition on the form of $\mathcal{T}$, which guarantees chase termination, and finally we estimate the complexity of computing the chase. Due to space constraints we provide here only proof sketches. Full proofs can be found in the extended version of the paper available at https://arxiv.org/abs/2002.05064

Theorem 1 (Termination of Transactions is Undecidable)**.** *It is undecidable whether there is a terminating chase sequence for a list $s = \langle\, model, queue\,\rangle$ wrt a document theory $\mathcal{T}$.*

Proof Sketch. The theorem is proved by a reduction of the halting problem for Turing machines. Given a Turing machine M, we define a document theory $\mathcal{T}$, which encodes M. The theory $\mathcal{T}$ contains axioms, which specify a single document form and a field used for storing the content of the tape of M, and axioms for daemons, which encode transitions of M. Then we define a list $initqueue$ of instructions, which encode the first symbols of the initial configuration of M and enforce execution of a transaction, which launches a transition of M from the initial state. Then it can be shown that there is a terminating chase sequence for $\langle\, \langle\,\rangle, initqueue\,\rangle$ iff M halts. $\square$

In fact, the form of the theory $\mathcal{T}$ used in the theorem shows that non-termination may be caused by the possibility to change a field value of

the same document or execute the same transaction infinitely many times. The definition of $SetFieldTrigger$ and $ExecTrans$ functions in $\mathcal{T}$ admits cyclic references between instructions and transactions. In the following, we observe that if one forbids cycles then chase termination is guaranteed.

Definition 2 (Dependency Graph). *A dependency graph over a document theory $\mathcal{T}$ is a directed graph with the set of vertices V equal to $Form$ $Names \times (FieldNames \cup TransNames \cup \{CreateDoc\})$ and the set of edges E defined as follows.*

For any $(form, name), (form', name') \in V$, there is an edge from $(form, name)$ to $(form', name')$ if there is $[\theta, q]$ in the definition of $SetFieldTrigger$ or $ExecTrans$ functions in $\mathcal{T}$, in which θ is a $(form, name)$-condition and $q = \mathtt{conc(queue, instr)}$, for a list $\mathtt{queue}$ and queue extension $\mathtt{instr}$, such that there is a $(form', name')$-instruction in the definition of $\mathtt{instr}$.

Definition 3 (Locally Simple Document Theory). *A document theory $\mathcal{T}$ is called locally simple if the dependency graph over $\mathcal{T}$ is acyclic.*

Theorem 2 (Local Simplicity Implies Termination of Transactions). *For any locally simple document theory $\mathcal{T}$ and constant lists $\mathtt{model}$, $\mathtt{queue}$, there is a terminating chase sequence for $\langle\, \mathtt{model}, \mathtt{queue}\, \rangle$ wrt $\mathcal{T}$.*

Proof Sketch. We show that for any such $\mathtt{model}$ and $\mathtt{queue}$, there is a finite chase sequence $s_0, \ldots, s_n$, where $s_0 = \langle\, \mathtt{model}, \mathtt{queue}\, \rangle$, $n \geqslant 1$, such that $s_n = \langle\, model', \mathtt{tail(queue)}\, \rangle$, where $|model'| = |\mathtt{model}| + p$, for some $p \geqslant 0$. This yields that any instruction from $\mathtt{queue}$ can be processed in a finite number of steps, from which the claim follows. $\square$

Although the theorem states that local simplicity guarantees termination, it does not provide any insight on how difficult it is to compute the effects of transactions. The next result indicates that the complexity is high, which is due to the possibility to create exponentially many documents by using recursive instruction terms. For $n \geqslant 0$, let $\mathtt{1exp}(n)$ be the notation for 2^n and for $k \geqslant 1$, let $(\mathtt{k+1})\mathtt{exp}(n) = 2^{\mathtt{kexp}(n)}$.

Theorem 3 (Computing Effects of Transactions is Hard). *For any $k \geqslant 1$, $n \geqslant 0$, there exists a locally simple document theory $\mathcal{T}$ and a constant list $\mathtt{queue}$, both of sizes linear in k, n, such that the terminating chase sequence for $s_0 = \langle\, \langle\, \rangle, \mathtt{queue}\, \rangle$ wrt $\mathcal{T}$ has the form $s_0, \ldots, s_m$, where $m \geqslant \mathtt{kexp}(n)$ and $s_m = \langle\, model, \langle\, \rangle\, \rangle$, for a list $model$ such that $|model| \geqslant \mathtt{kexp}(n)$.*

Finally, let us formulate a sufficient condition, which guarantees polynomial boundedness of effects of transactions. Let G be a dependency graph over a document theory $\mathcal{T}$ and for $form \in FormNames$, $name \in FieldNames \cup TransNames \cup \{CreateDoc\}$, let s be a $(form, name)$-instruction in a queue extension from the definition of $SetFieldTrigger$

or $ExecTrans$ functions in $\mathcal{T}$. We call the term s *document generating* if either $name = CreateDoc$ or $s = \texttt{Rec}[\langle\ \rangle, h, \texttt{DocFilter}]$, where $h = \texttt{conc}(\langle\ \langle\ form, CreateDoc\ \rangle\ \rangle, g(\alpha))$, or $(form, name)$ has a successor vertex $(form', name')$ in G, which is given by a document generating term.

Theorem 4 (Polynomially Bounded Effects of Transactions)**.** *Let $\mathcal{T}$ be a locally simple document theory such that in any queue extension from the definition of $SetFieldTrigger$ or $ExecTrans$ functions in $\mathcal{T}$, there are no document generating* $\texttt{Rec}$*-terms.*

Then for any constant list $\texttt{model}$ *and a list of instructions* $\texttt{queue}$*, the terminating chase sequence for $s_0 = \langle\ \texttt{model}, \texttt{queue}\ \rangle$ has the form $s_0, \ldots, s_n$, where n is exponentially bounded by the size of $\mathcal{T}, s_0$, and $s_n = \langle\ model', \langle\ \rangle\ \rangle$, for a list $model'$ of size polynomially bounded by the size of $\mathcal{T}$ and s_0.*

Proof Sketch. Let N be the maximal number of instruction terms in a queue extension from the definition of $SetFieldTrigger$ or $ExecTrans$ functions in $\mathcal{T}$. Clearly, N is bounded by the size of $\mathcal{T}$. Let $\texttt{model}$, $\texttt{queue}$ be lists, which satisfy the conditions of the lemma, and let k be the rank of instruction $t = \texttt{head(queue)}$ wrt $\mathcal{T}, \texttt{model}$. By definition, k is bounded by the number of vertices in the dependency graph over $\mathcal{T}$ and thus, it is bounded by the size of $\mathcal{T}$. We show by induction on k that there is a chase sequence $s_0, s_1, \ldots, s_n$, such that $s_0 = \langle\ \texttt{model}, \texttt{queue}\ \rangle$, $s_n = \langle\ model', \texttt{tail(queue)}\ \rangle$, $|model'| \leqslant |\texttt{model}| + N$, and $n \leqslant (N \cdot (|model| + N))^k$. Then there is a terminating chase sequence $s_0, \ldots, s_m$ for s_0, where $s_m = \langle\ m, \langle\ \rangle\ \rangle$, $|m| \leqslant |\texttt{queue}| \cdot (|\texttt{model}| + N)$, and $m \leqslant |\texttt{queue}| \cdot (N \cdot (|model| + N))^k$, which proves the theorem. $\square$

5. Conclusions

We have shown that document theories (and thus, the Document Modeling approach) implement a Turing-complete computation model even in the presence of a tractable language of arithmetic operations (over document field values) and queries (for selecting collections of documents). This confirms that one of the main sources of the computational complexity are the definitions of daemons, which specify transactions and relationships between them. If the definitions are given in a way that allows for executing the same transaction or changing the value of a document field infinitely many times, then it is possible to implement computations of any Turing machine. We have shown that disallowing cyclic relationships between transactions guarantees decidability of transaction termination (importantly, cycles can be easily detected by a syntactic analysis of axioms of a document theory), but the complexity of computing effects of transactions even in this case is high, if creating documents in loops is possible. In fact, using looping in transactions is natural, since it allows for performing updates over collections of documents. If documents can be

only modified in loops, but not created, then the complexity of computing effects of transactions is decreased and we have noted a case when the effects are polynomially bounded. In further research, we plan to make a more detailed complexity analysis for various (practical) restrictions on the definition of daemons. In this paper, we did not study the contribution of query languages to the complexity of computing effects of transactions and we have adopted a relatively simple query language. Since daemons employ document queries to modify collections of documents, it would be important to study the interplay between these two sources of complexity.

References

1. Ershov Yu.L., Goncharov S.S., Sviridenko D.I. Semantic Programming. *Information processing 86: Proc. IFIP 10th World Comput. Congress.* 1986, vol. 10, Elsevier Sci., Dublin, pp. 1093-1100.

2. Ershov Yu.L., Goncharov S.S., Sviridenko D.I. Semantic Foundations of Programming. *Fundamentals of Computation Theory: Proc. Intern. Conf.* FCT 87, Kazan, 116–122. Lect. Notes Comp. Sci., 1987, vol. 278. https://doi.org/10.1007/3-540-18740-5_28

3. Goncharov S.S., Sviridenko D.I. Σ-programming. Transl. II. *Amer. Math. Soc.*, 1989, no. 142, pp. 101-121.

4. Goncharov S.S., Sviridenko D.I. Σ-programming and its Semantics. *Vychisl. Systemy*, 1987, no. 120, pp. 24-51. (in Russian).

5. Goncharov S.S., Sviridenko D.I. Theoretical Aspects of Σ-programming. *Lect. Notes Comp. Sci.*, 1986, vol. 215, pp. 169-179. https://doi.org/10.1007/3-540-16444-8_13

6. Goncharov S.S. Conditional Terms in Semantic Programming. *Siberian Mathematical Journal*, 2017, vol. 58, no. 5, pp. 794-800. https://doi.org/10.1134/S0037446617050068

7. Goncharov S.S., Sviridenko D.I. The Logic Language of Polynomial Computability. *Doklady Mathematics*, 2019, vol. 99, no.2, pp. 11-14.

8. Goncharov S.S., Sviridenko D.I. Recursive Terms in Semantic Programming. *Siberian Mathematical Journal*, 2018, vol. 59, no. 6, pp. 1279-1290.

9. Kazakov I.A., Kustova I.A., Lazebnikova E.N., Mantsivoda A.V. Building locally simple models: theory and practice. *The Bulletin of Irkutsk State University. Series Mathematics*, 2017, vol. 21, pp. 71-89. (in Russian). https://doi.org/10.26516/1997-7670.2017.22.71

10. Malykh A.A., Mantsivoda A.V. Document modeling. *The Bulletin of Irkutsk State University. Series Mathematics*, 2017, vol. 21, pp. 89-107. (in Russian). https://doi.org/10.26516/1997-7670.2017.21.89

11. Mantsivoda A.V., Ponomaryov D.K. A Formalization of Document Models with Semantic Modelling. *Bulletin of Irkutsk State University, Series Mathematics*, 2019. vol. 27, pp. 36-54.

12. Ospichev S., Ponomarev D. On the Complexity of Formulas in Semantic Programming. *Siberian Electronic Mathematical Reports*, 2018, vol. 15, pp. 987-995.

13. Reiter R. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems.* MIT Press, 2001. https://doi.org/10.7551/mitpress/4074.001.0001

14. Vityaev E.E. Semantic Probablistic Inference of Predictions. *The Bulletin of Irkutsk State University. Series Mathematics*, 2017, vol. 21, pp. 33-50. (in Russian).

Andrei Mantsivoda, Doctor of Sciences (Physics and Mathematics), Professor, Irkutsk State University, 1, K. Marx st., Irkutsk, 664003, Russian Federation; Sobolev Institute of Mathematics SB RAS, 4, Koptyug pr., Novosibirsk, 630090, Russian Federation, tel.: +7 (3952) 521241, e-mail: `andrei@baikal.ru`

Denis Ponomaryov, Candidate of Sciences (Physics and Mathematics), Sobolev Institute of Mathematics SB RAS, 4, Koptyug pr., Novosibirsk, 630090, Russian Federation; Ershov Institute of Informatics Systems SB RAS, 6, Lavrentyev pr., Novosibirsk, 630090, Russian Federation; Novosibirsk State University, 1, Pirogov st., Novosibirsk, 630090, Russian Federation, tel.: +7 (383) 3306660, e-mail: `ponom@iis.nsk.su`

О завершаемости транзакций над семантическими документными моделями

А. В. Манцивода[1,2], Д. К. Пономарев[2,3,4]

[1] *Иркутский государственный университет, Иркутск, Российская Федерация*

[2] *Институт математики им. С. Л. Соболева СО РАН, Новосибирск, Российская Федерация,*

[3] *Институт систем информатики им. А. П. Ершова СО РАН, Новосибирск, Российская Федерация*

[4] *Новосибирский государственный университет, Новосибирск, Российская Федерация*

Аннотация. Рассматривается парадигма документного моделирования, которая дает формальную основу для работы с документами в системах управленческого планирования. В данной работе документные модели формулируются в рамках логического формализма – языка семантического моделирования – и рассматривается проблема распознавания завершаемости транзакций, заданных документной моделью, для любого возможного входа. Показывается, что в общем случае данная проблема алгоритмически неразрешима. Формулируются достаточные условия, гарантирующие разрешимость и полиномиальную ограниченность результата выполнения транзакций.

Ключевые слова: семантическое моделирование, документная модель, транзакция, цепь.

Список литературы

1. Ershov Yu. L., Goncharov S. S., Sviridenko D. I. Semantic Programming // Information processing 86: Proc. IFIP 10th World Comput. Congress. Vol. 10. Elsevier Sci., Dublin, 1986. P. 1093–1100.

2. Ershov Yu. L., Goncharov S. S., Sviridenko D. I. Semantic Foundations of Programming // Fundamentals of Computation Theory: Proc. Intern. Conf. FCT 87, Kazan, 116–122. Lect. Notes Comp. Sci. 1987. Vol. 278. https://doi.org/10.1007/3-540-18740-5_28

3. Goncharov S. S., Sviridenko D. I. Σ-programming, Transl. II. // Amer. Math. Soc. 1989. N 142. P. 101–121.

4. Goncharov S. S., Sviridenko D. I. Σ-programming and its Semantics// Vychisl. Systemy. 1987. N 120. P. 24–51. (in Russian).

5. Goncharov S. S., Sviridenko D. I. Theoretical Aspects of Σ-programming // Lect. Notes Comp. Sci. 1986. Vol. 215. P. 169–179. https://doi.org/10.1007/3-540-16444-8_13

6. Goncharov S. S. Conditional Terms in Semantic Programming // Siberian Mathematical Journal. 2017. Vol. 58, N 5. P. 794–800. https://doi.org/10.1134/S0037446617050068

7. Гончаров С.С., Свириденко Д.И. Логический язык описания полиномиальной вычислимости // Доклады РАН. 2019. Т. 485, № 1. С. 11–14.

8. Гончаров С. С., Свириденко Д. И. Recursive Terms in Semantic Programming // СМЖ. 2018. Т. 59, № 6. С. 1279–1290.

9. Построение локально-простых моделей: методология и практика / И. А. Казаков, И. А. Кустова, Е. Н. Лазебникова, А. В. Манцивода // Известия Иркутского государственного университета. Сер. Математика. 2017. Т. 22. С. 71–89. https://doi.org/10.26516/1997-7670.2017.22.71

10. Малых А. А., Манцивода А. В. Документное моделирование // Известия Иркутского государственного университета. Сер. Математика. 2017. Т. 21. С. 89–107. https://doi.org/10.26516/1997-7670.2017.21.89

11. Mantsivoda A. V., Ponomaryov D. K. A Formalization of Document Models with Semantic Modelling // The Bulletin of Irkutsk State University. Series Mathematics. 2019. Vol. 27, pp. 36–54.

12. Ospichev S., Ponomarev D. On the Complexity of Formulas in Semantic Programming // Siberian Electronic Mathematical Reports. 2018. Vol. 15. P. 987–995.

13. Reiter R. Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems. MIT Press, 2001. https://doi.org/10.7551/mitpress/4074.001.0001

14. Витяев Е. Е. Семантический вероятностный вывод предсказаний // Известия Иркутского государственного университета. Сер. Математика. 2017. Т. 21. С. 33–50.

Андрей Валерьевич Манцивода, доктор физико-математических наук, профессор, Институт математики, экономики и информатики, Иркутский государственный университет, Российская Федерация, 664003, г. Иркутск, ул. К. Маркса, 1; Институт математики им. С. Л. Соболева СО РАН, Российская Федерация, 630090, г. Новосибирск, пр. Академика Коптюга, 4, тел.: +7 (3952) 521241, e-mail: andrei@baikal.ru

Денис Константинович Пономарев, кандидат физико-математических наук, Институт математики им. С. Л. Соболева СО РАН, Российская Федерация, 630090, г. Новосибирск, пр. Академика Коптюга, 4; Институт систем информатики им. А. П. Ершова СО РАН, Российская Федерация, 630090, г. Новосибирск, пр. Лаврентьева, 6; Новосибирский государственный университет, Российская Федерация, 630090, г. Новосибирск, ул. Пирогова, 1, тел.: +7 (383) 3306660, e-mail: ponom@iis.nsk.su