Серия «Математика»
2019. Т. 27. С. 36—54

Онлайн-доступ к журналу:
http://mathizv.isu.ru

И З В Е С Т И Я
Иркутского
государственного
университета

УДК 510.62:004.82
MSC 68T27, 68N19
DOI https://doi.org/10.26516/1997-7670.2019.27.36

A Formalization of Document Models with Semantic Modelling*

A. V. Mantsivoda
Irkutsk State University, Irkutsk, Russian Federation,
Sobolev Institute of Mathematics, Novosibirsk, Russian Federation

D. K. Ponomaryov
Ershov Institute of Informatics Systems, Sobolev Institute of Mathematics, Novosibirsk State University, Novosibirsk, Russian Federation

Abstract. In this paper, we formalize the general concept of a document model in terms of the Semantic Modelling (SM) paradigm. We argue that the idea of using documents as a basic metaphor for modelling appears to be very useful, since it provides a balance between the logical tools for knowledge processing and cognitive aspects for a much wider audience than the community of professional mathematicians. A subject domain can be arbitrarily complex by its nature, but humans tend to choose those primitives, which are convenient for cognition. The notion of a document is an example of such a primitive, which has been employed for centuries and clearly remains topical in the era of information systems. The significant outcome of constructing the semantics of document models within the SM paradigm is that Semantic Modelling makes document models executable. Executable models can be directly used as practical information systems, and this feature makes the programming stage unnecessary. Replacing programming with modelling has a great impact on the efficiency of IT systems development and maintenance, and makes these systems friendly for the Artificial Intelligence tools.

Keywords: semantic modelling, document model

1. Introduction

Semantic Modelling is an area of mathematical logic and Artificial Intelligence, which describes domains by representing them in a logical form.

* The research was supported by the Russian Science Foundation (Grant No. 17-11-01176)

Semantic models can be used by knowledge mining systems, robots, automated reasoners, and machine learning algorithms. Several years ago, one of the well-known AI inventions – neural networks and Deep Learning - made a significant breakthrough in information processing. We are convinced that now it is time for the Semantic Modelling to contribute.

The use of semantic models is able to provide a breakthrough in the automation of business management. Today, SAP R/3, Oracle ERP, Microsoft Dynamic Ax, Salesforce, and similar systems reign here. By and large, they are all coded programs that run business processes.

However programming has an extremely unpleasant flaw, which we call the *lobotomy of meanings*. The holistic and beautiful semantics of interacting business processes is cut into pieces and dissolved in numerous program modules and databases. The semantics, which is dissolved in programs, is not directly accessible to us, as well as to the AI tools.

Based on our own experience, we are convinced that today business management is ready for replacing programming by Semantic Modelling. Properly constructed models can be operational, like information systems, and this makes the programming phase unnecessary. Unlike programs, semantic models retain the meaning of business processes and are open to the AI tools and robots.

2. Modelling vs Programming

We believe that now it is time for the beginning of a gentle, but inevitable process, which will replace programming by modelling in many significant areas. Semantic Modelling is a mature and very developed area of research, it provides a number of important advantages, since

- skipping the "programming phase" makes an application development several times faster and fundamentally reduces the costs of maintenance and modernization;
- modelling can supply AI and robots with explicit domain semantics (whereas in programs the semantics dissolves);
- models (as opposed to programs) are directly comprehensible by a much wider range of specialists, e.g., managers, consultants, analysts, and lawyers.

A traditional scenario for developing an IT system starts with composing technical specifications for programmers. A semantic model can be also interpreted as a formal technical specification. But as soon as such a model is built, programming is no longer required, since the model itself is "alive" and executable.

We are developing and implementing the bSystem, a platform-as-a-service, which supports the technology of building executable semantic models as tools for constructing industrial-level information systems. bSystem is based on document modelling, a version of Semantic Modelling built around a concept of a document. bSystem relies on a paradigm of the Semantic Modelling, which has been developed by Yu. Ershov, S.Goncharov and D.Sviridenko [1]– [6]. They used the concept of a list superstructure over datatype models in order to declaratively describe domains of discourse in such a way that the resulting models could be executable. This means that data and knowledge handling procedures can be automatically extracted from a model.

The idea of using documents as a basic metaphor for modelling (which was coined in [8]) appeared to be very fruitful. The complexity of different subject domains can not be addressed without appealing to cognitive aspects. A subject domain can be arbitrarily complex by its nature, but humans tend to choose those primitives, which are convenient for cognition. The notion of a document is an example of such a primitive, which has been employed for centuries and is evidently topical in the context of information systems. It would be fair to say that this concept has been formed by the human experience in information structuring and organization of social processes. A document has a static and dynamic nature. The static aspect of a document defines its structure and content, while the dynamic one corresponds to modifications and versioning. Numerous activities, e.g., those related to the Enterprise Resource Planning and similar areas, can be naturally described in terms of document processing, including the static and dynamic aspects of documents. In this context, it is important to study the complexity of formalization of these aspects from the cognitive and computational point of view.

Theoretically, the key task in implementing this approach is to formalize the life cycle of a semantic model (in particular, the life cycle of a document model). In practice, models evolve over time and constantly change their contents and descriptions. It is also essential that the models are open systems, subject to the influence of external parties (oracles). To reflect these features in document modelling, we develop and formalize in this paper the concept of the life cycle of document models. It is based on the notion of a model fixed point. A fixed point is a stable state, which the model tries to reach as a reaction to external disturbances. When some external oracle causes disturbances in the model (for example, by introducing new data), the model becomes unstable, perhaps even incorrect. By reaching a new fixed point, the model reconciles with the updates.

Thus, reaching fixed points allows the model to find a balance between soundness and dynamic updating. Only at the fixed point the model does guarantee its soundness. In case, when a model cannot reach a new fixed point after an external intervention, it considers this intervention as invalid and restores the previous state it had before. This, in particular, allows the model to cope with erroneous and malicious behaviours of oracles. In this way, static logical models can be efficiently used in the dynamically changing world.

The procedural behaviour of semantic models allows us to replace programming in IT projects. Within the bSystem, we applied the technology to the development of heavily interactive Web services in a FINTEC company, for HR management, enterprise budgeting, retailer business analytics, and other real-life applications. Our practice shows that replacing programming by modelling results in at least five-fold increase in efficiency, both in development, and maintenance and modernization, with the corresponding savings of financial, labour, and time resources.

3. Principles of Document Modelling

A document model is a version of an executable semantic model, which is based on the metaphor of the document as the basic construct for logical descriptions. Document modelling implements our concept of knowledge management.

First, the document models are executable. This makes the programming stage unnecessary, since the model itself can play the role of an information system.

Second, the model uses the notion of a *document* as a metaphor. The document model is organized as a collection of logical structures that can be interpreted as "ordinary" documents, while preserving all the advantages of the Semantic Modelling and Artificial Intelligence. On the other hand, from the users perspective, working with this model is identical to the conventional work with documents.

In this section, we recall the basic notions of document modelling from [7;8].

Let Ω be a set. A *sequence* (over Ω) is an expression

$$(e_1, \dots, e_m),$$

where $e_i \in \Omega$ are some elements. The empty sequence with no elements is denoted as $(\,)$.

To determine the number of elements in a sequence, we use the notion of *cardinality*. We define the following cardinalities:

– $(\,)$ is the empty sequence
– ? is a sequence with zero or one element
– ! is a sequence containing exactly one element
– $+$ is a sequence with one or more elements
– $*$ is a sequence with any number of elements.

A countable set of constants

$$▯ = \{n_1, n_2, \ldots\}$$

is called the set of *names* (identifiers). This set is divided into two disjoint subsets of document *form names* $▯_F$ and document *field names* $▯_D$.

A *document field description* is a tuple

$$\langle d, \mathbf{c} \rangle,$$

where $d \in ▯_D$ is a field name and $\mathbf{c}$ is a cardinality (of the field value). For instance, $\langle age, ! \rangle$ is an example of a field description.

A *document field* is a pair

$$\langle d, v \rangle$$

where $d \in ▯_D$ is a field name and v is a sequence, a field value.

A document is the main concept of a document model.

– A document consists of *fields* and has a unique *id*, which is an identifier given by a natural number.
– Each document is of one of the predefined *document forms*, which play the role of templates that describe the structure of admissible documents and transactions.
– Transactions represent instructions, which must be executed upon creation of a new document of a specific form, or a value change for some field in a document of a specific form.

An *instruction* is an expression of the form $CreateDoc(f)$ or $SetField(id, d, v)$, and a *condition* is an expression $CreateDoc(f)$, $SetField(f, d)$, or $SetField(f, d, v)$, where $f \in ▯_F$ is a document form name, *id* is a document identifier, $d \in ▯_D$ is a field name, and v is a field value. Informally, $CreateDoc(f)$ stands for the event of creating a document of a certain form f, while $SetField(f, d, v)$ means that the value of a field with name d is changed in a document of a form f to a value v. The expression $SetField(id, d, v)$ is interpreted similarly. It is assumed that the form name, document id, and field name parameters can be values of a document field.

Further, we introduce concepts of a rule and document form and the central notion of a document. In comparison with [7; 8], we use a simplified definition of a rule in this paper for clarity of presentation.

A rule is an expression of the form

$$G \to P$$

where G is a condition expression (a premise) and P is a sequence of instructions (a conclusion).

A document form is a tuple

$$\mathbf{f} = \langle\, f, \{\mathbf{d}_1, \ldots, \mathbf{d}_n\}, \{\mathbf{r}_1 \ldots \mathbf{r}_k\} \rangle$$

where $f \in \mathbb{I}_F$ is a form name, $\{\mathbf{d}_1, \ldots, \mathbf{d}_n\}$ is a finite set of field descriptions, and $\{\mathbf{r}_1 \ldots \mathbf{r}_k\}$ is a finite set of rules such that:

– every condition appears in at most one rule as a premise;
– f is the only form name parameter, which can occur in a premise.

A *document* is a tuple

$$\mathbf{o} = \langle\, \mathbf{f}, id, \{\mathbb{d}_1, \ldots, \mathbb{d}_n\}\rangle$$

where $\mathbf{f} \in \mathbb{I}_F$ is a document form name, id is a document identifier, and $\{\mathbb{d}_1, \ldots, \mathbb{d}_n\}$ is a finite set of document fields such that their names are exactly the names from the field descriptions of $\mathbf{f}$.

A *document model* is a tuple consisting of a domain of field values and a finite set of document forms

$$\mathcal{D} = \langle \Omega, \{\mathbf{f}_1, \ldots \mathbf{f}_m\}\rangle$$

A *state* of a document model $\mathcal{M}$ is a finite set of documents

$$\langle\{\mathbf{o}_1, \ldots \mathbf{o}_n\}\rangle$$

where every $\mathbf{o}_i$ is a document of a form $\mathbf{f}_j$, where $j \in \{1, \ldots, m\}$, with fields having values from Ω.

Therefore, the sets of field descriptions given in document forms represent the static part of the document model (i.e., the structure of documents). The dynamic part is given by rules. They have the following informal semantics: whenever the condition in the premise of a rule holds (i.e., if a document of a specific form is created or a value of a specific field in a document is changed), the instructions from the conclusion of the rule must be executed. The collection of instructions, which must be executed upon rule firing, is called *transaction.*

In the document model approach proposed in [8] it is assumed that actions like $CreateDoc$ and $SetField$ can be performed by oracles (external information sources, e.g., users) and these actions can yield rule firing. If any of instructions fails during rule firing, then the whole transaction fails. Thus, rules determine possible transitions between document model states, since rule firing can yield different collections of documents (or document versions). The operators, which check whether rule conditions are met and which generate a sequence of instructions to be executed, are called *daemons* (similar to the notion used in process programming).

It should be clear from the informal definition above that firing a rule may cause other rules to fire and this process can repeat indefinitely long in general. If there are no rules that can fire and no instructions that must be executed, then the document model state is *stable.* In practice, only a document model in a stable state can be used for querying, because otherwise the information in a document model may be incomplete, due to some pending instructions. Therefore, the principle task is to be able to compute stable states of a given document model wrt inputs of oracles.

4. Fundamentals of Semantic Modelling

The concept of Semantic Modelling [1] – [5] has been introduced in 80's as an alternative to the logic programming paradigm. It is a formalism, which is powerful enough to abstract away from the implementation details of programming, while keeping the possibility to manually specify the order of computations. One of the important features, which makes Semantic Modelling particularly convenient is the built-in list type, which is general enough to naturally represent numerous data types occurring in practice. An algorithm or a problem domain is described in the Semantic Modelling as a *model*, a logical theory, which contains two basic subtheories. The first one specifies the list type and computability over lists. The second one introduces (possibly recursive) definitions of domain specific predicates, whose extension one wants to compute. The semantics of models (in particular, the extension of predicates) is given in terms of fixed-points. The language of Semantic Modelling is expressive enough to reduce all common program verification tasks to logical entailment. In fact, the language is so powerful that it allows for reasoning over the constructed models both, at the syntactic and semantic level.

The language of the Semantic Modelling is a first-order language with equality interpreted over first-order list structures. Their domain is the set of all possible lists (including the distinguished empty list $[\,]$) over some set Ω and their signature contains basic operations for working with lists. In particular, the domain contains all elements from Ω, as well as lists with elements from Ω, lists of lists, and so on. If every $a_1, \ldots, a_n$ is an element from Ω or a list, then $[a_1, \ldots, a_n]$ denotes a list over $a_1, \ldots, a_n$. A list is *plain* if it is empty or it holds that $a_i \in \Omega$, for all $i = 1, \ldots, n$.

There are basic list operations such as

- *head*, which gives the first element of the list, if the list is not empty, and $[\,]$ otherwise;
- *tail*, which gives the remaining part of the list with the first element removed, if the list is not empty, and $[\,]$ otherwise;
- *cons* which appends an element to a list as the first one;
- *conc*, which concatenates two lists.

There is a predicate $\sqsubseteq$ such that for any lists α, β, it holds $\alpha \sqsubseteq \beta$ if α is the starting segment of β and a predicate $\in$, which designates containment in a list.

The following axioms for list operations are assumed to be included into any logical theory in the language of the Semantics Modelling:

$tail(cons(\alpha,\beta)) = \beta$
$head(cons(\alpha,\beta)) = \alpha$
$(\alpha \neq [\,] \to cons(head(\alpha), tail(\alpha)) = \alpha$
$tail([\,]) = [\,] \quad head([\,]) = [\,]$
$cons(conc(\alpha,\beta),\gamma) = conc(\alpha, cons(\beta,\gamma))$
$conc(conc(\alpha,\beta),\gamma) = conc(\alpha, conc(\beta,\gamma))$
$conc([\,],\alpha) = conc(\alpha,[\,]) = \alpha$

There are also list induction axioms of the form

$$([\Phi]^x_{[\,]} \wedge \forall\alpha\forall\beta([\Phi]^x_\alpha \to [\Phi]^x_{cons(\alpha,\beta)})) \to \Phi$$

where Φ is an arbitrary formula and $[\Phi]^x_t$ denotes the formula obtained by substituting free occurrences of the variable x with the term t (avoiding variable collisions).

There is a foundation axiom

$$(\forall\delta\ \forall\gamma \in \delta\ \ \Phi(\gamma) \to \Phi(\delta)) \to \forall\alpha\Phi(\alpha)$$

and the equal-size axiom of the form

$$\alpha = \beta\ \equiv\ \forall\gamma \in \alpha\ \gamma \sqsubseteq \beta \wedge (\gamma \neq \beta \to \\ \exists\delta \in \alpha\ cons(\gamma,\delta) \sqsubseteq \alpha \wedge cons(\gamma,\delta) \sqsubseteq \beta)))$$

where the bounded quantifier of the form $\gamma \in \alpha$ means that γ is an element of the list α. Bounded quantifiers play an important role in the Semantic Modelling, since in the descriptions of real-world domains the choice of elements to reason over is typically finite. Formulas with bounded quantifiers correspond to the class of Δ_0-formulas. The computational complexity of these formulas over lists has been studied in [9].

There is also a Δ_0-determination axiom, which states the existence of a table function for any list α, which selects only those elements from α, for which some property Φ holds. Finally, there is a Δ_0-selection axiom, which we also omit here for brevity; an interested reader can find the details in [1] – [5]. The mentioned axioms constitute the background of any logical theory in the language of the Semantic Modelling and are denoted as `GES` (the Goncharov-Ershov-Sviridenko theory).

5. Formalization of Document Models

We are now ready to provide a formalization of document models in the framework of the Semantic Modelling.

Let $\mathcal{D} = \langle\Omega, \{\mathbf{f}_1, \dots \mathbf{f}_m\}\rangle$ be a document model. We define a logical theory $\mathcal{T}_\mathcal{D}$ for $\mathcal{D}$ as a theory in signature Σ', where Σ' consists of predicate and function symbols introduced in the formulas below. In particular, Σ' contains pairwise disjoint subsets of constants $FieldNames = \{fieldName_1, \dots, fieldName_m\}$, $FormNames = \{formName_1, \dots, formName_n\}$, and $Vals = \{v_1, \dots, v_k\}$, $m, n, k \geq 1$, which stand for the field and form names in $\mathcal{D}$, as well as the names for the elements of Ω. It also contains function *Form*, which gives a form name for a document, and the distinguished constants *CreateDoc* and *SetField*, which are used to represent instructions.

We use the following notions and modelling conventions in our formalization.

– All the above mentioned constants are interpreted as distinct elements, i.e., there is the unique name assumption axiom in $\mathcal{T}_{\mathcal{D}}$ for these constants.
– An *instruction* is given as a list of the form $[CreateDoc, formName]$ or $[SetField, docID, fieldName, value]$, where $formName \in FormNames$, $fieldName \in FieldNames$, $docID$ represents a natural number, and $value$ is a field value.
– Natural numbers are modelled as lists in a standard way and we assume that there is the sum $+$ function and $>$ predicate in Σ' (we also use the shortcut $\leq$), which are expressible in terms of the list operations.
– A *queue* is a list of instructions to be executed. A queue is updated by daemons, which implement triggers on the events such as creating a new document (of a given form) or changing a field value in a given document.
– A *situation* is a list of instructions, it will represent the history of executed instructions in our formalization. Moreover, the last executed instruction will appear first in a situation.
– A *field* is given as a list with two elements: $[fieldName, v]$, where $fieldName \in FieldNames$ and v is a plain list over $Vals$, a value for a field. The default value for any field is the empty list $[\,]$.
– A *document* is a list of fields (the order of fields in the list is arbitrary).
– A *model* is a list consisting of triples $[docID, doc, sit]$, where $docID$ is a natural number, doc is a document, and sit is a situation. A model stores a document version in each situation which has ever taken place. The head of this list is a triple, whose situation is the current one, i.e., this situation consists of instructions (a history) that have yielded the model. We use the following notation, where M is a model: $History(M) = tail(tail(head(M)))$.

In the formulas below, we assume that all the free variables are universally quantified.

We begin with the axioms of the theory $\mathcal{T}_{\mathcal{D}}$, which represent the static part of the document model $\mathcal{D}$, i.e., field descriptions and document forms.

For every field description $\langle fieldName, card\rangle$ from a document form in $\mathcal{D}$, the theory $\mathcal{T}_{\mathcal{D}}$ contains an equation of the form

$$Field(x) = fieldName \equiv head(x) = fieldName \wedge Card(tail(x)) \quad (5.1)$$

where $fieldName \in FieldNames$ and $Card$ is a predicate, which represents the cardinality of the elements in a plain list x. One can easily show the following property of the theory GES: for any list x and any cardinality $card \in \{?, !, +, *\}$, there is a predicate $Card$ such that $\mathtt{GES} \cup \{Card\} \models Card(x)$ iff x is a plain list of cardinality $card$.

To represent document forms, $\mathcal{T}_{\mathcal{D}}$ contains the following equation, which defines how a blank document of a particular form must look like:

$$\begin{aligned} &Blank(name) = document \equiv \\ &\quad (\bigwedge_{f \in FormNames} name \neq f \ \wedge \ document = [\,]) \ \vee \bigvee_{f \in FormNames} \varphi_f \quad (5.2) \end{aligned}$$

where every φ_f is a conjunction of the form

$$\begin{aligned} name = f \wedge \exists x_1, \ldots, x_n \in document \bigwedge_{i=1\ldots n} Field(x_i) = fieldName_i \wedge \\ \forall x \in document \ (x = x_1 \vee \ldots \vee x = x_n) \wedge tail(x) = [\,] \wedge \\ Form(document) = name \end{aligned}$$

specifying, which fields must be present in a document of a form f (note that all the field values are set to default, i.e., to the empty list).

To formulate the dynamic part of the document model, we introduce three auxiliary functions. The following function (which is defined recursively) gives the last used ID for a document in a model. When evaluated on a given model and $id = 0$, the definition of the function implements search for a triple in a *model*, whose first element (the ID) is the greatest one among all other triples in *model*.

$$\begin{aligned} GetLastDocID(model, id) = docID \equiv model = [\,] \wedge docID = id \vee \\ (\ head(head(model)) > id \wedge \\ docID = GetLastDocID(tail(model), head(head(model)))\ ) \vee \\ head(head(model)) \leq id \wedge docID = GetLastDocID(tail(model), id) \end{aligned} \quad (5.3)$$

The next function gives an actual version of a document (from a model) by its ID. It implements search for the first triple with a given ID (contained in a model) and outputs the found document. If no triple with a given ID is present in the model, then the function returns the empty list.

$$\begin{aligned} &GetDocByID(docID, model) = document \equiv \\ &head(head(model)) = docID \wedge document = head(tail(head(model))) \vee \\ &\quad head(head(model)) = [\,] \wedge document = [\,] \vee \\ &\quad head(head(model)) \neq docID \wedge head(head(model)) \neq [\,] \wedge \\ &\qquad document = GetDocByID(docID, tail(model)) \end{aligned} \tag{5.4}$$

The next function provides a field value from the actual (the last) version of a document with a given ID:

$$\begin{aligned} &GetFieldValue(docID, fieldName) = tail((head(tail(\\ &\quad FindFieldPosition(fieldName, [[\,], GetDocByID(docID)]) \;))) \end{aligned} \tag{5.5}$$

It relies on the recursive function *FindFieldPosition*, which “splits” a given document into a paritioned one (denoted as *pdocument* below), which has the form $[list_1, list_2]$ such that $conc(list_1, list_2) = document$ and $head($ $list_2)$ is a field with the required name (if there exists one in a document). This auxiliary function is employed further to implement the operation of changing a field value in an existing document.

$$\begin{aligned} &FindFieldPosition(fieldName, pdocument) = pdocument' \equiv \\ &head(head(tail(pdocument))) = fieldName \wedge pdocument' = pdocument \vee \\ &\quad tail(pdocument) = [\,] \wedge pdocument' = pdocument \vee \\ &head(head(tail(pdocument))) \neq fieldName \wedge tail(pdocument) \neq [\,] \wedge \\ &\quad pdocument' = FindFieldPosition(fieldName, \texttt{newpdocument}) \end{aligned} \tag{5.6}$$

where `newpdocument` is a shortcut for

$$[cons(\; head(tail(pdocument)), head(pdocument)\;),\; tail(tail(pdocument))]$$

In other words, if $pdocument = [[a_1, \ldots, a_m], [b_1, b_2, \ldots, b_n]]$, then it holds that $newpdocument = [[a_1, \ldots, a_m, b1], [b_2, \ldots, b_n]]$.

Now we are ready to define the main recursive operator, which implements the dynamic part of the document model given by transactions. Given a queue, it updates a model and the queue to the new state, based on the definition of daemons. For the sake of readability, we split the definition of the operator into three formulas (combined with disjunction) and comment on them separately.

First of all, if the queue is empty (there is nothing to do), then the update of a model is trivial, i.e., nothing is changed. If an instruction in the queue is not a valid one (either *CreateDoc* or *SetField*) then the whole queue is skipped and the model returned by the *Update* operator is the initial model (this implements the transaction mechanism adopted in the document modelling approach):

$$\begin{aligned}
&\mathit{Update}(\mathit{initialmodel}, \mathit{model}, \mathit{queue}) = \mathit{model}' \equiv \\
&\qquad \mathit{queue} = [\,] \wedge \mathit{model}' = \mathit{model} \vee \\
&\qquad (\mathit{queue} \neq [\,] \wedge (\mathit{head}(\mathit{head}(\mathit{queue})) \neq \mathit{CreateDoc}) \wedge \\
&\qquad (\mathit{head}(\mathit{head}(\mathit{queue})) \neq \mathit{SetField}) \wedge \mathit{model}' = \mathit{initialmodel}\ ) \vee
\end{aligned} \tag{5.7}$$

Otherwise the queue contains an instruction to create a document of a specific form, or change a field value in a document having a certain ID. In the first case, a blank document is created (which is implemented by using existenial quantification), the instruction is removed from the queue, and the queue is extended by daemons, which are implemented by the *NewDocTrigger* function (see further). Finally, the created document is added to the model and the *Update* operator is evaluated recursively on fresh inputs. If a blank document of a form with name $formName$ can not be created (due to the fact that $formName \notin FormNames$), then the queue is skipped and *Update* returns the initial model:

$$\begin{aligned}
&\mathit{head}(\mathit{head}(\mathit{queue})) = \mathit{CreateDoc} \wedge \\
&\qquad \exists \mathit{document}\ \ \mathit{document} = \mathit{Blank}(\texttt{formName}) \wedge \\
&\qquad ((\mathit{document} = [\,] \wedge \mathit{model}' = \mathit{initialmodel}) \ \vee (\mathit{document} \neq [\,] \wedge \\
&\mathit{model}' = \mathit{Update}(\mathit{initialmodel}, \mathit{cons}(\texttt{newdoc}, \mathit{model}), \texttt{extendedQueue})))\vee
\end{aligned} \tag{5.8}$$

where `formName` stands for $\mathit{tail}(\mathit{head}(\mathit{queue}))$, `newdoc` is an abbreviation for the list of the form $[\mathit{GetLastDocID}(\mathit{model}, 0) + 1, \mathit{document}, \texttt{newhistory}]$, `newhistory` is a shortcut for $\mathit{cons}([\mathit{CreateDoc}, \texttt{formName}], \mathit{History}(\mathit{model}))$, and `extendedQueue` is $\mathit{NewDocTrigger}(formName, \mathit{tail}(\mathit{queue}))$.

In other words, `newdoc` represents a fresh document to be created in the model and `newhistory` is a history extended with an action of creating a document. The expression `extendedQueue` will be a queue obtained by firing a document creation trigger.

The case of *SetField* instruction in the queue is treated similarly, but the formalization is technically more complex, because modifying an existing document takes more steps than creating a fresh one:

$$\begin{aligned}&head(head(queue)) = SetField \wedge \\ &\qquad ((tail(\texttt{pdocument}) = [\,] \wedge model' = initialmodel) \vee \\ &\qquad\qquad (tail(\texttt{pdocument}) \neq [\,] \wedge model' = \\ &Update(initialmodel, conc(\texttt{updatedDoc}, model), \texttt{extendedQueue})) \;) \quad (5.9)\end{aligned}$$

where docID, fieldName, newFieldValue stand for $head(tail(head(queue)))$, $head(tail(tail(head(queue))))$, and $head(tail(tail(tail(head(queue)))))$, respectively (recall the instruction modelling conventions listed in the beginning of this section). Further, pdocument is an abbreviation for $FindFieldPosition($ fieldName, $GetDocByID(\texttt{docID}, model))$ and updatedDoc is a shortcut for

$$\begin{aligned}&conc(cons([\texttt{fieldName}, \texttt{newFieldValue}], head(\texttt{pdocument})), \\ &\qquad\qquad\qquad\qquad tail(tail(\texttt{pdocument})))\end{aligned}$$

Finally, extendedQueue is a shortcut for $ChangedFieldTrigger($ $GetDocByID(docID, model), fieldName, newFieldValue, tail(queue))$.

Thus, updatedDoc is a document with an updated field value and 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 a specified name in a given document. Note that in this case it holds $tail(\texttt{pdocument}) = [\,]$ by the definition of $FindFieldPosition$ function.

Now we define functions, which implement daemons. Their purpose is to extend the queue with a sequence of instructions depending on whether a new document is created or a field value in an existing document is changed. There are also additional conditions, which influence, what sequence of instruction is chosen. Both functions have similar definitions:

$$\begin{aligned}&(NewDocTrigger(formName, queue) = queue') \equiv \Phi \\ &\qquad (ChangedFieldTrigger(doc, fieldName, fieldValue, queue) = \\ &\qquad\qquad = queue') \equiv \Psi \quad (5.10)\end{aligned}$$

where Φ and Ψ are formulas of the form

$$\begin{aligned}&\bigvee_{i\in I}(Condition_i \wedge queue' = queue_i) \vee \\ &\qquad\qquad \vee\, (\bigwedge_{i\in I} \neg Condition_i) \wedge queue' = queue_{else}\end{aligned}$$

where I is a finite index set and every $Condition_i$ in Φ, is a formula of the form $formName = name$, where $name \in FormNames$, while every $Condition_i$ in Ψ is of the form $Form(document) = name_1 \wedge fieldName = name_2$ or $formName = name_1 \wedge fieldName = name_2 \wedge fieldValue = value$, where $name_1 \in FormNames$, $name_2 \in FieldNames$. Moreover, none of the formulas $Condition_i$ appear twice in Φ or Ψ. Finally, $queue_i$ and $queue_{else}$ are formulas of the form $cons(instr_1, .., cons(instr_n, queue), ...)$, where $n \geq 0$ (if $n = 0$ then we assume that $queue_i/queue_{else}$ is $queue$) and every $instr_k$, for $k = 1, \ldots, n$, is an expression of the form $[CreateDoc, formName]$ or $[SetField, docID, fieldName, fieldValue]$, with the parameters defined recursively by the following grammar (we use the Backus-Naur form here for simplicity of presentation):

fieldValue ::= *value*

formName ::= *formname* | GetFieldValue(docID, fieldName)

docID ::= *number* | GetFieldValue(docID, fieldName)

fieldName ::= *fieldname* | GetFieldValue(docID, fieldName)

where $value \in Vals$, $formname \in FormNames$, *number* (a list representing a natural number), and $fieldname \in FieldNames$ are constants. Notice that the order of instructions appended to *queue* is determined by the form of expressions $queue_i$ and $queue_{else}$.

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

6. Properties of Formalization

The theory $\mathtt{GES} \cup \mathcal{T}_{\mathcal{D}}$ is a conservative extension of $\mathtt{GES}$, i.e., the axioms of $\mathcal{T}_{\mathcal{D}}$ do not modify the semantics of the predicates and functions introduced in the background theory of the Semantic Modelling, which can be proved by showing that any model of $\mathtt{GES} \cup \mathcal{T}_{\mathcal{D}}$ can be expanded to a model of $\mathtt{GES}$.

For a document model $\mathcal{D}$, a state of $\mathcal{D}$ is represented by a list (appearing as *model* in the axioms of $\mathcal{T}_{\mathcal{D}}$), which consists of triples $[docID, document, situation]$ (see the conventions and notations in the previous section). Similarly, a sequence of instructions is represented as a *queue* list. Thus, we say that a list *model* or *queue* represents a state of $\mathcal{D}$ or a sequence of instructions, respectively. These lists are used as arguments in the definition of the *Update* operator in $\mathcal{T}_{\mathcal{D}}$. Transactions over a document model are formalized in the scope of $NewDocTrigger$ and $ChangedFieldTrigger$ functions defined in $\mathcal{T}_{\mathcal{D}}$. We note that the definition of these functions recalls the “if then else” construct and can be represented by using conditional terms in the formalism of the Semantic Modelling [6]. By the definition of *Update* operator, if some instruction issued by a rule can not be executed (e.g., an attempt to create a document of an unknown form or to modify a non-existing field in a document is made) then the whole queue of instructions is ignored and the model returned by *Update* is the initial model (being the first parameter of the operator). This implements the transaction mechanism adopted in the document modelling, which cancels the whole bunch of instructions if at least one of them fails.

Let $\mathcal{D}$ be a document model, S a state of $\mathcal{D}$, Q a sequence of instructions, and let the lists *model* and *queue* represent S and Q, respectively. The formalization of the document model $\mathcal{D}$ provided by $\mathcal{T}_{\mathcal{D}}$ has the following key properties. It holds that

$$\mathtt{GES} \cup \mathcal{T}_{\mathcal{D}} \models \mathit{Update}(\mathit{model}, \mathit{model}, \mathit{queue}) = \mathit{model}'$$

iff model' represents a stable model state obtained from S by applying Q. In particular, it is possible to determine a state obtained from the empty model (with no documents) by executing a set of user-specified instructions. For example, *queue* can contain instructions representing user actions in the system (e.g., creating several blank documents and changing some field values in them). Then the value of $\mathit{Update}([\ ], [\ ], \mathit{queue})$ gives the model state obtained by taking into account daemons, which implement rule firing on the user input.

An even more important observation is that the projection and planning problems [10; 11] can be reduced to entailment from $\mathtt{GES} \cup \mathcal{T}_{\mathcal{D}}$. In the case of projection, assume one is interested whether a document of a form with name $formName$ will exist in a state obtained from some model state S by applying a sequence of instructions Q (for simplicity, we assume below that S and Q are the lists, which represent the model state and instructions, respectively). Then it suffices to check whether the following holds:

$$\begin{aligned} \mathtt{GES} \cup \mathcal{T}_{\mathcal{D}} \models\ & \exists \mathit{document} \in \mathit{model} \\ & \quad \mathit{Form}(\mathit{document}) = formName \ \wedge\ \mathit{model} = \mathit{Update}(S, S, Q) \end{aligned}$$

Similarly, the planning problem is reduced to entailment. Assume one wants to know whether there is a sequence of instructions Q, which brings a given model state S to a desired one, where, e.g., a document of a form with name $formName$ exists. Then it suffices to verify whether

$$\begin{aligned} \mathtt{GES} \cup \mathcal{T}_{\mathcal{D}} \models\ & \exists Q \ \ \exists \mathit{document} \in \mathit{model} \\ & \quad \mathit{Form}(\mathit{document}) = formName \ \wedge\ \mathit{model} = \mathit{Update}(S, S, Q) \end{aligned}$$

If the proof of this entailment is constructive, then it is possible to extract the corresponding sequence of instructions from the proof, e.g., a value for the variable Q.

Since the property of being a finite list is expressible in the formalism of Semantic Modelling, we also obtain the following important result saying that it is possible to verify termination of rule firing via logical entailment. There exists a formula Φ such that $\mathtt{GES} \cup \mathcal{T}_{\mathcal{D}} \models \Phi$ iff the state obtained from a model state S by executing a sequence of instructions Q is finite.

7. Summary and Outlook

We have introduced a formalization of document models in terms of Semantic Modelling. Specifically, we have formulated the static part of a document model (i.e., descriptions of fields and document forms) as appropriate list structures. The dynamic part (transactions and daemons) is given in terms of recursive predicates, whose extension is computed via fixed points within the approach of Semantic Modelling. We have noted that the proposed formalization is expressive enough for solving the key meta-level problems in the document modelling such as projection, planning, and termination of transactions. In particular, this allows for finding modelling errors, simulating the consequences of user actions in a system without changing its content, and finding a sequence of actions, which brings a system to a desired state. In practice, this allows for answering questions like: what actions must be performed in order to get an item on retail stock. These features go beyond the capabilities of today's document workflow processing and Enterprise Resource Planning systems.

We plan to investigate the concept of the local simplicity [8] in terms of our formalization. Also the algorithmic complexity of verifying the existence of finite fixed points w.r.t. rule firing will be studied, as well as restrictions that must be imposed on the form of the business processes (daemons) in order to guarantee an efficient solution to the projection and planning problems. A quite interesting and promising direction is the integration of the proposed formalism with the semantic probabilistic inference [12] in order to solve AI tasks over document models. These are the topics for the further research.

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, Lect. Notes Comp. Sci., 1987, vol. 278. pp. 116-122. 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. 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
8. Malykh A.A., Mantsivoda A.V. Document Modelling. *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
9. Ospichev S., Ponomarev D. On the Complexity of Formulas in Semantic Programming. *Siberian Electronic Mathematical Reports*, 2018, vol. 15, pp. 987–995.
10. Pirri F. and Reiter R. Some Contributions to the Metatheory of the Situation Calculus. *J. ACM*, 1999, vol. 46, no. 3, pp. 325–364.
11. 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
12. E. Vityev. 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, tel.: (3952)521241 Sobolev Institute of Mathematics, 4, Acad. Koptyug av., Novosibirsk, 630090, Russian Federation, tel.: +7 (383) 3306660 (e-mail: `andrei@baikal.ru`)

Denis Ponomaryov, Candidate of Sciences (Physics and Mathematics), Ershov Institute of Informatics Systems, 6, Acad. Lavrentjev pr., Novosibirsk, 630090, Russian Federation; Sobolev Institute of Mathematics, 4, Acad. Koptyug av., Novosibirsk, 630090, Russian Federation, tel.: +7 (383) 3306660; Novosibirsk State University, 1, Pirogova str., Novosibirsk, 630090, Russian Federation (e-mail: ponom@iis.nsk.su)

Received 10.10.18

Формализация документных моделей средствами семантического моделирования

А. В. Манцивода

Иркутский государственный университет, Иркутск, Российская Федерация, Институт математики им. С. Л. Соболева СО РАН, Новосибирск, Российская Федерация

Д. К. Пономарев

Институт систем информатики им. А.П. Ершова СО РАН, Институт математики им. С. Л. Соболева СО РАН, Новосибирский государственный университет, Новосибирск, Российская Федерация

Аннотация. В данной работе мы формализуем общую концепцию документной модели в терминах семантического моделирования (СМ). Мы считаем, что идея использования документов в качестве базовой метафоры для моделирования является очень полезной, поскольку она обеспечивает необходимый баланс между логическими методами обработки знаний и когнитивными аспектами, связанными с доступностью данных методов для намного более широкой аудитории, чем сообщество профессиональных математиков. Предметная область может быть сколь угодно сложной по своей природе, однако люди склонны выбирать такие описательные примитивы, которые более удобны для восприятия. Понятие документа является примером такого примитива, который веками использовался людьми, и остается одним из важнейших в эпоху информационных систем. Ключевым качеством описания семантики документных моделей в рамках парадигмы СМ является то, что семантическое моделирование делает документные модели исполняемыми. Исполняемые модели могут напрямую использоваться в качестве практических информационных систем, и эта черта делает этап программирования излишним. Заменяя программирование на моделирование, мы принципиально повышаем эффективность разработки и поддержки информационных систем, а также делаем такие системы дружелюбными для средств искусственного интеллекта.

Ключевые слова: семантическое моделирование, документная модель.

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, Lect. Notes Comp. Sci. Kazan, 1987. Vol. 278. P. 116–122. 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.
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. Построение локально-простых моделей: методология и практика / И. А. Казаков, И. А. Кустова, Е. Н. Лазебникова, А. В. Манцивода // Изв. Иркут. гос. ун-та. Сер. Математика. 2017. Т. 22. С. 71-89. https://doi.org/10.26516/1997-7670.2017.22.71

8. Малых А. А., Манцивода А. В. Документное моделирование // Изв. Иркут. гос. ун-та. Сер. Математика. 2017. Т. 21. С. 89-107. https://doi.org/10.26516/1997-7670.2017.21.89
9. Ospichev S., Ponomarev D. On the Complexity of Formulas in Semantic Programming // Siberian Electronic Mathematical Reports. 2018. Vol. 15. P. 987—995.
10. Pirri F., Reiter R. Some Contributions to the Metatheory of the Situation Calculus // J. ACM. 1999. Vol. 46, N 3. P. 325—364.
11. 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
12. Витяев Е. Е. Семантический вероятностный вывод предсказаний // Изв. Иркут. гос. ун-та. 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.: (3952)521241 (e-mail: andrei@baikal.ru)

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

Поступила в редакцию 10.10.18