Відмова від Шаблонів і Перезапис Термів

Для моделювання робочих бізнес-процесів в Maude

Bartosz Zieliński
Факультет комп'ютерних наук
Університет Лодзі
Поморська 149/153, 90-236 Лодзь, Польща
bzielinski@uni.lodz.pl

Paweł Maślanka
Факультет комп'ютерних наук
Університет Лодзі
Поморська 149/153, 90-236 Лодзь, Польща
pmaslan@uni.lodz.pl

Cancellation Patterns and Term Rewriting

For Business Workflow Modeling in Maude

Bartosz Zieliński
Department of Computer Science
University of Łódź
Pomorska 149/153, 90-236 Łódź, Poland
bzielinski@uni.lodz.pl

Paweł Maślanka
Department of Computer Science
University of Łódź
Pomorska 149/153, 90-236 Łódź, Poland
bzielinski@uni.lodz.pl

***Анотація*—Зміна рангу процесів, на відміну від більшості інших моделей робочих процесів, не локальна справа. Хоча концептуально задача проста, вона, як відомо, важко моделюється з Петрі мережами - це математичний апарат, найбільш часто використовується для забезпечення семантики Workflow конструкції. В роботі показано, що використання перезапису, настільки ж ефективне, як мережі Петрі для моделювання паралельних і розподілених систем, з екваціональними специфікаціями і використовується з не локальними змінами.**

***Abstract*—Cancellation of a range of activities is, unlike most of the other workflow patterns, a non-local affair. While conceptually simple, it is notoriously difficult to model with Petri Nets – a mathematical formalism most commonly used to provide semantics to workflow constructs. Here we demonstrate that combining term rewriting, equally useful as Petri Nets to model paralell and distributed systems, with equational specifications allows dealing naturally with non-local changes. To this end, we implement in the term rewriting system Maude a toy workflow simulation framework (essentially based on token passing, similarly as in the case of Petri Nets) and then augment it with a YAWL-like cancellation region construct..**

Ключові слова— термін перезапису; бізнес-процес; робочий процес; Maude

Keywords— term rewriting; business process; workflow; Maude;

I. INTRODUCTION

Cancellation of a range of activities [1] is, unlike most of the other workflow patterns, except perhaps for a XOR split [2], a non-local affair. While conceptually simple, it is notoriously difficult [1] (though not impossible) to model with Petri Nets – a mathematical formalism most commonly used to provide semantics to workflow constructs. It requires simultanous removal of tokens from an arbitrary subset of places by a cancelling transition, and thus may require an exponential increase of complexity of the net. Using naturally distributed formalisms such as pi-calculus to model workflows (as in [3]) leads to another kind of problem with modeling cancellation patterns. Namely, the issue is with *simultanous* cancelling of a set of activities, which is obviously difficult in any system based on message passing.

Here we demonstrate that combining term rewriting, equally useful as Petri Nets to model paralell and distributed systems, with equational specifications allows dealing naturally with non-local changes. To this end we implement in the term rewriting system Maude a toy workflow simulation framework (based on token passing, similarly as in the case of Petri Nets) and then augment it with a YAWL-like cancellation region construct [1].

II. PRELIMINARIES ON MAUDE AND TERM REWRITING

Maude [4] is a language and execution system based on term rewriting [5], [6] and many sorted equational logic [7]. In

this section we recall some basic mathematical definitions to fix the notation and to make the presentation self-contained.

An algebraic signature $\Sigma=(\Sigma_S, \Sigma_F)$ consists of a finite poset of sorts Σ_S (sorts are like type names, and the order corresponds to subtyping) and a finite set of function signatures Σ_F . Each function signature is of the form $f : s_1 s_2 \dots s_n \rightarrow s$, where f is a function symbol and s_i's are sorts in Σ_S Symbols c such that $c : \rightarrow s \in \Sigma_F$ are called *constants* of sort s..

A Σ-algebra A is an assignment of a set $[[s]]_A$ to each sort s in Σ_S and a function (interpretation) $[[f]]_A : [[s_1]]_A \times \dots \times [[s_n]]_A \rightarrow [[s]]_A$ to each function $f : s_1 s_2 \dots s_n \rightarrow s$. It is required that $[[s_1]]_A \subseteq [[s_2]]_A$ whenever $s_1 \leq s_2$. We call x an element of A *if* $x \in [[s]]_A$ for some $s \in \Sigma_S$.

Let $V:=\{V_s | s \in \Sigma_S\}$ be a collection of sets where elements of V_s are variables of sort s. A term algebra $T_\Sigma(V)$ has "sort-safe" terms as elements and function symbols interpreted by themselves. We denote by T_Σ the algebra of ground Σ-terms.

In Maude, the poset of sorts Σ_S in the signature is implicitly augmented with separate top elements $[K]$ (referred to as kinds) for connected components K of Σ_S. We denote $[s]:=[K]$ for any $s \in K$. Giving a function symbol arguments of a wrong kind is treated as a syntax error. Giving a function symbol arguments of a wrong sort (but correct kind) is legal, but makes the result member of a kind but not of a sort.

Maude supports the so-called mixfix syntax - underscores in the function name correspond to consecutive arguments. Thus if Σ_F contains $\_+\_ : Nat\ Nat \rightarrow Nat$ and $\_?\_ : Bool\ Nat\ Nat \rightarrow Nat$ we can use expressions such as 1+2 or *false ? 1 : 2*.

Maude allows to define Σ-algebras as quotients of term algebras by a congruence generated by a pair (A,E) of sets of equalities. Equalities in A, specified as attributes on function symbols, define certain structural properties of binary operators such as associativity, commutativity, idempotence or unitality. E consists of actual conditional equalities interpreted as directed simplification rules. It is assumed that simplifications terminate and each term has the unique canonical (irreducible) form modulo properties from A. Maude represents equivalence classes in $T_\Sigma /\equiv_{A,E}$ with irreducible elements of $T_\Sigma /\equiv_A$. Conditions in conditional equalities are conjunctions of unconditional equalities and membership axioms.

Reductions with respect to equalities can be viewed as computations of values and as such they represent denotational aspect of the problem. The behavioural aspect is represented through rewritings. Single step rewrites are defined with a set of conditional rewriting rules of the form $T_1 \Rightarrow T_2$ *if* C, where C is a conjunction of equalities, membership axioms and rewriting axioms. A rewriting axiom $K_1 \Rightarrow K_2$ is satisfied if K_1 can be rewritten in one or more steps to K_2 .

A rewriting system consists of a signature, a set of equalities and equational attributes as well as a collection of rewriting rules. Definitions of rewriting systems are collected in Maude modules, either functional (which cannot include rewriting rules and simply define quotient algebras) or system ones which can include rewriting rules. Modules can be parametrized by theories which define properties of classes of systems with which the module can be instantiated.

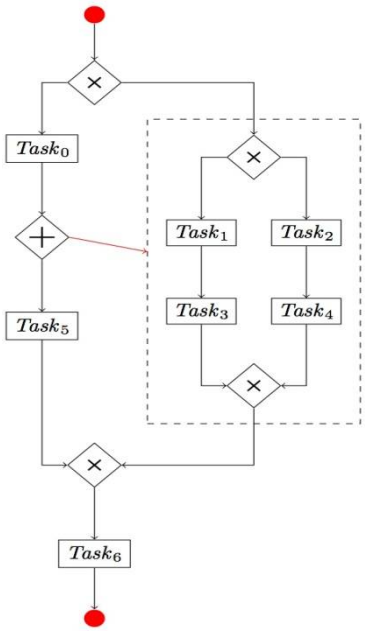

Fig. 1. Example workflow with cancellation region

III. CANCELLATION REGIONS

Cancellation refers to aborting activations of a selection of tasks within the workflow in response to some event, activity or decision by the user. As an example consider the process of preparing the travel in which separate threads of activities devoted to booking a hotel, airplane ticket, etc., might need to be simultaneously aborted when the user changes plans. An abstract example is presented in Figure 1. There, when the red path of execution is taken at the XOR decision gate, the region of the workflow enclosed in dashed lines gets cancelled, which means that whichever of the four tasks are active (we have four possibilities, e.g., {Task1, Task2}, {Task1, Task4}, etc.) becomes inactive. In case of workflow system based on Petri Nets this might be implemented through removing tokens from places inside the cancelled region, which requires special inhibitor edges (see. e.g., [1]), or an additional subnet with complicated topology. In what follows we will demonstrate how to implement cancellation within rewriting models for workflow systems. We will show this within a context of a toy rewriting theory for a workflow presented in the next section. However, it should be noted that the method is completely general.

IV. A TOY REWRITING THEORY FOR WORKFLOWS

Our toy rewriting theory will allow us to model workflows with the following elements: tasks (no distinction is made between kinds of tasks, such as user, automatic, manual, etc.), XOR and AND joins and splits, and, finally, end nodes and the single start node. Late we augment the formalism with cancellation regions.

We start by defining syntax. We represent state of the workflow as an ensamble of sort *Config* of workflow items of sort *Item* collected together using the associative and commutative (empty) operator __ . The relevant definitions are as follows:

```
sorts  Item Config .
subsort Item < Config .
op nil : -> Config .
op __ : Config Config -> Config
        [assoc comm id: nil] .
```

Note that due to the subsort declaration a single *Item* is also a *Config*. Note also the neutral element *nil* of the "__" operator. Now we define the constructors for various *Item*s constituting the state of the workflow. Each such item (except for the start node) is identified by an identifier of sort *Id* and has a state of sort *State*. Possible states of items are "ready", "active" and "next". Identifiers can be given as values of a predefined sort Qid (quoted identifiers) consiting of strings of charactes which are valid Maude identifiers preceded by a single quotation mark (e.g., 'q). Later we will also define additional identifier constructors. All other attributes of items, unique to each kind of item (and distinguishing between different kinds of them) are gathered together as terms of sort *Part*. All items, except for the start node are constructed as triples of identifier, state, and ``the rest'' with the constructor [_,_,_]:

```
sorts Id State Part Op. subsort Qid < Id.
ops ready active next : -> State .
ops xor and : -> Op .
op start : State Id -> Item .
op [_,_,_] : Id State Part -> Item  .
ops conn task :  Id -> Part .
op end : -> Part .
ops split : Id Id Op -> Part .
ops join  : Id Op -> Part .
```

We define six kinds of items: the start node and the remaining five kinds of nodes distinguished by *Part*s: end items, task, connector (conn), and, finally, split and join gates. It is assumed that tasks, gates, start and end nodes connect together with connectors. Connectors have one auxilliary attribute: the identifier of the next workflow object which the connector connects to and which is to be activated by the given connection object. The usage of the explicit connector objects simplifies the rules, as now items of other kinds are activated by and activate only connectors. This allows us to avoid the combinatorial explosion caused by considering each combination of types of consecutive items.

The start node does not have its own identifier (it is unique), it does, however, have state, as well as the identifier of the unique connector to be activated next. End nodes have no auxilliary attributes. Each task node contains identifier of the unique connector to be activated next. Both join and split gates contain identifiers of their operation of sort *Op* (either "and" or "xor"). In addition, each join (resp. split) gate contains the unique identifier of the next connector (resp. the identifiers of two next connectors.

V. REWRITE RULES OF A TOY MODEL

The rewrite rules define the operational semantics of our toy workflow model and allow it to be actually executed (for instance to test its correctness). The general idea is as follows: There are three states the workflow can be in: "ready", "active" and "next", although only tasks pass through all three. The remaining components use only "ready" and "next". The ready component can pass into "next" or "active" states when its input connectors are in the "next" state. A given component can make all or some of its output connectors transition to the "next" state when it is itself in the "next" state. The additional "active" state of tasks simulates the task being executed (perhaps for an extended amount of time). The rules are as follows. First we define variables to use in the rules:

```
vars X X' Y Y' Z : Id . var P : Part .
var S : State . var O : Op .
```

Then come the rewriting rules. For example, the following rules allows any task to transition by itself from the active state into the "next" state:

```
rl [X, active, task(Y)]
            => [X, next, task(Y)] .
```

The following rules describe the activation of tasks by an input connector and the activation of an output connector by a task:

```
rl [X, next, conn(Y)] [Y, ready, task(Z)]
    => [X, ready, conn(Y)] [Y, active,
          task(Z)] .
rl [X, next, task(Y)] [Y, ready, conn(Z)]
     => [X, ready, task(Y)] [Y, next,
           conn(Z)] .
```

As the last example consider the following rules which deal with activation of output connectors of split gates:

```
rl [Z, next, split(X, Y, and)] [X, ready,
   conn(X')] [Y, ready, conn(Y')] =>
     [Z, ready, split(X, Y, and)]
[X, next, conn(X')] [Y, next, conn(Y')] .
rl [Z, next, split(X, Y, xor)] [X, ready,
conn(X')] => [Z, ready, split(X, Y, xor)]
[X, next, conn(X')] .
rl [Z, next, split(X, Y, xor)] [Y, ready,
conn(Y')] => [Z, ready, split(X, Y, xor)]
[Y, next, conn(Y')] .
```

The END split activates both of its output rules. In case of a XOR split gate there are two rules, one for the activation of each of the output connectors. Thus, a XOR split may activate either one of its two outputs, depending on which of these rules gets chosen by the system. We omit the rest of the rules as they are similar.

VI. IMPLEMENTATION OF THE CANCELLATION REGIONS

We implement the cancellation regions utilizing the fact that Maude (except for frewrite strategy) will attempt to reduce term to the canonical form after each rewrite step. Thus, we can implement cancellation through equational reduction, which, from the point of view of rewriting, happens immediately. More precisely, we introduce a special workflow item which holds the list of identifiers of workflow objects to be cancelled. Cancellation in this contex means changing the state of all objects in the cancellation region to "ready". The relevant code is described below.

First we need operators and sorts for lists of identifiers:

```
sort IdList . subsort Id < IdList .
op nil : -> IdList .
op __ : IdList IdList -> IdList
        [assoc id: nil] .
```

The following operator represents cancellation regions in the workflow:

```
op cancel : Id IdList IdList -> Item .
```

The first argument is the identifier of the cancellation region. The remaining two are the list of identifiers of the objects within the region and the list of identifiers of workflow objects the activity of which currently remains to be cancelled. When the cancellation region is inactive the value of this second list is *nil*. In fact, both lists do not contain identifiers of connectors which are treated separately. The cancellation region can be activated through the connector object as defined with the following rule:

```
rl [X, next, conn(Y)] cancel(Y, IL, nil)
=> [X, ready, conn(Y)] cancel(Y, IL, IL).
```

Note that this rule applies only if the last argument of *cancel* is equal to nil. The main workhorse is the following equation:

```
eq cancel(X, IL, Y IL') [Y, S, P] =
    cancel(X, IL, IL') [Y, ready, P]
         ccon(Y, cnt(P)) .
```

Note that the start node cannot be the part of cancellation region. All other workflow objects in our toy model are matched by the pattern [Y, S, P]. As the lists of identifiers in *cancel* are not supposed to contain connectors' identifiers we have to deal with them separately. To this end, the equality above introduces a workflow item constructed with operator `ccon : Id Nat -> Item` which is supposed to de-activate all input connectors for workflow object Y:

```
eq [X, S, conn(Y)] ccon(Y, s N) = [X,
ready, conn(Y)] ccon(Y, N) .
eq ccon(Y, 0) = nil .
```

The second argument of *ccon* is the number of input connectors which remains to be de-activated. Its initial value is the number of incoming connectors computed for each workflow item type using the function:

```
op cnt : Part -> Nat .
eq cnt(join(X, O)) = 2 .
eq cnt(conn(X)) = 0 .
eq cnt(P) = 1 [owise] .
```

The *owise* attribute means ``in all other cases". Thus, the number of incoming connectors is 2 for join gates, 0 for connectors, and 1 in all other cases.

VII. Example

Consider the example of a workflow in Fig. 1. It can be represented in our formalism as a term

```
start(next, c(0)) ['e1,ready,end]
['g1,ready,split(c(1), c(2), and)]
['g2,ready,split(c(4), c(5), xor)]
['g3,ready,join(c(7), and)]
cancel('c1, 'g4 'g5 't1 't2 't3 't4, nil)
*** etc.
```

Here we have used a separate constructor for connector identifiers: `op c : Nat -> Id .` After some analysis, one sees hat the workflow is flawed. Indeed, if the cancellation region is invoked, then at least the left one of the branches of the last AND join gate is never invoked, and so the end node is never activated. We can easily confirm that by searching (with the search command) for irreducible (with respect to rewritings) terms reachable from the start state in which end is not activated (c is assigned to the initial workflow description):

```
search c =>! C:Config ['e1, ready, end] .
```

and verifying that Maude indeed finds several solutions.

VIII. Conclusion

We have presented a toy rewriting model for workflows and we have extended it with cancellation regions implemented through equations. While the toy model is not very useful in itself, the method for extending models with cancellations is very general and does not depend on the details of our toy model. It shows that cancellation patterns can be naturally implemented within the framework of rewriting systems.

References

[1] Van Der Aalst, Wil MP, and Arthur HM Ter Hofstede. "YAWL: yet another workflow language." Information systems 30.4 (2005): 245-275.

[2] van Der Aalst, Wil MP, et al. "Workflow patterns." Distributed and parallel databases 14.1 (2003): 5-51.

[3] Puhlmann, Frank, and Mathias Weske. "Using the π-calculus for formalizing workflow patterns." International Conference on Business Process Management. Springer Berlin Heidelberg, 2005.

[4] Clavel, Manuel, et al. "The maude 2.0 system." International Conference on Rewriting Techniques and Applications. Springer Berlin Heidelberg, 2003.

[5] Meseguer, José. "Conditional rewriting logic as a unified model of concurrency." Theoretical computer science 96.1 (1992): 73-155.

[6] Meseguer, José, and Grigore Roşu. "The rewriting logic semantics project." Theoretical Computer Science 373.3 (2007): 213-237.

[7] Meseguer, José. "Membership algebra as a logical framework for equational specification." Recent Trends in Algebraic Development Techniques (1998): 18-61.

[8] Clavel, Manuel, et al. All about maude-a high-performance logical framework: how to specify, program and verify systems in rewriting logic. Springer-Verlag, 2007.