

Logic for higher-order workflow of composite web services

Mihhail Matskin

School of Information and Communication
Technology at the Royal Institute of Technology,
KTH, P.O.Box Electrum 229, S-164 40 Kista,
Sweden

misha@imit.kth.se

telephone: +46 8 790 4128, fax: +46 8 7511793

Enn Tyugu

Estonian Business School,
Lauteri 3, 10114 Tallinn, Estonia;
Institute of Cybernetics
Akademia tee 21, 12618 Tallinn, Estonia

tyugu@ieee.org

telephone: +372 6204150, fax 3726204151

Abstract. We present logic that enables us to describe higher-order workflow precisely and to reason about the reachability of goals on workflow models. This is a suitable basis for business process analysis as well as for dynamic composition of services. Our approach extends workflow semantics and enables one to specify complex algorithms in terms of workflow. In particular, arbitrary cycles and hierarchical workflow become easily representable and obtain deeper semantics.

Keywords: higher-order workflow, web services, intuitionistic logic, service composition.

1. Introduction

Workflow methods that first appeared in business process modeling are being intensively used in web services now. Also the complexity of workflow management is increasing. Bearing in mind the perspective of dynamic composition of services, higher-order workflow is needed for representing web service models. This requires exact representation of workflow semantics. Petri nets have been the first choice in formalizing the workflow [7]. However, there are problems in applying higher-order Petri nets in workflow management—the conventional control structures are difficult to represent in this way.

We present here a logic that enables us to describe higher-order workflow precisely, and allows us to reason about the reachability of goals on workflow models. This is a suitable basis for dynamic composition of services. Our approach extends workflow semantics and enables one to specify complex algorithms in terms of workflow. In particular, arbitrary cycles and hierarchical workflow become easily representable and obtain deeper semantics. We are going to represent the logic on an example and try to avoid purely syntactic formal representation as much as possible.

2. Higher-order workflow

It is easy to represent partial order of tasks by a graph. Also various patterns of choice can be represented in this way. In particular, Petri nets are widely used for this purpose [7]. However, to be able to represent arbitrary control over execution of the tasks, one needs higher order features in the representation formalism, because the objects to be manipulated are the tasks themselves as we can see from the example below.

Let us take a process of buying a number of goods for a limited amount of some foreign currency as an example. First, we need a currency conversion task, let us call it *crmcConv* that converts a foreign currency *fCrmc* into the native currency *nCrmc*. Then we are going to use the tasks *choseProduct* and *addProduct* for making a choice of a new purchased product and adding it to the shopping cart. There has to be a task of finishing the process, e.g. giving the credit card number and authorizing the purchase, we call it *Finish*. As we have to repeat the tasks *choseProduct* and *addProduct* several times (the number of repetitions not known beforehand), we have to introduce a control over tasks that forms a cycle. In web service description languages and business process execution languages [1], [8] we have control constructs *sequence* and *cycle*

that vary from language to language to some extent, but can all be used in our case. We can represent the workflow now visually as a graph shown in Fig. 1. Looking at the Fig. 1 we see that there are two kinds of arcs. Thin lines express the ordering of tasks and are widely used in conventional workflow schemas. Thick lines express the control that the control nodes *cycle* and *sequence* have over the other tasks. Instead of *sequence* we could have used another control construct *bag* as well, because the ordering of tasks *chooseProduct* and *addProduct* is determined by the ordinary workflow anyway.

At a closer look one sees that Fig. 1 still does not show some important information about data passed between tasks and control constructs. Workflow languages used in web services include means for expressing data dependencies as well. One can extend a workflow graph by including explicit data nodes and use the arrows as input-output descriptions. When the order of execution of tasks is determined by data dependencies, one can drop the explicit ordering information. This is shown in Fig. 2, where the data entities: *fCrnc*, *nCrnc* etc. are explicitly shown together with their roles as inputs and outputs of tasks. We have introduced new data entities *spec*, *product*, *shoppingCart* and *receipt*. Their meaning should be rather obvious, except maybe of *spec* that is a specification for selecting goods to be bought, enabling us to use a generic task *chooseProd* for choosing the goods.

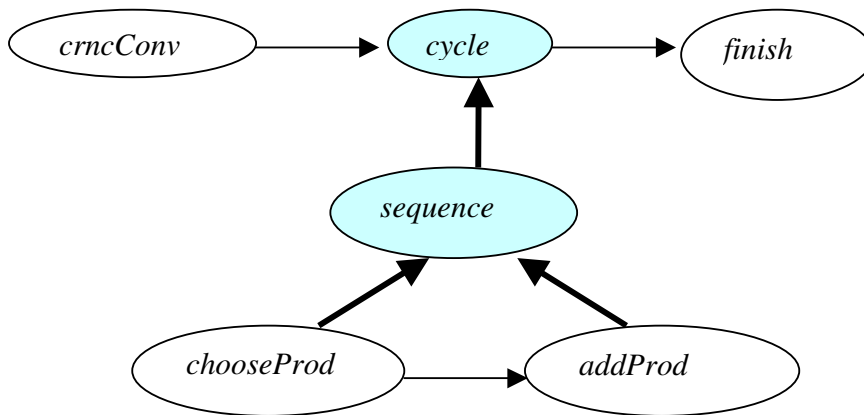


Figure 1. Higher-order workflow

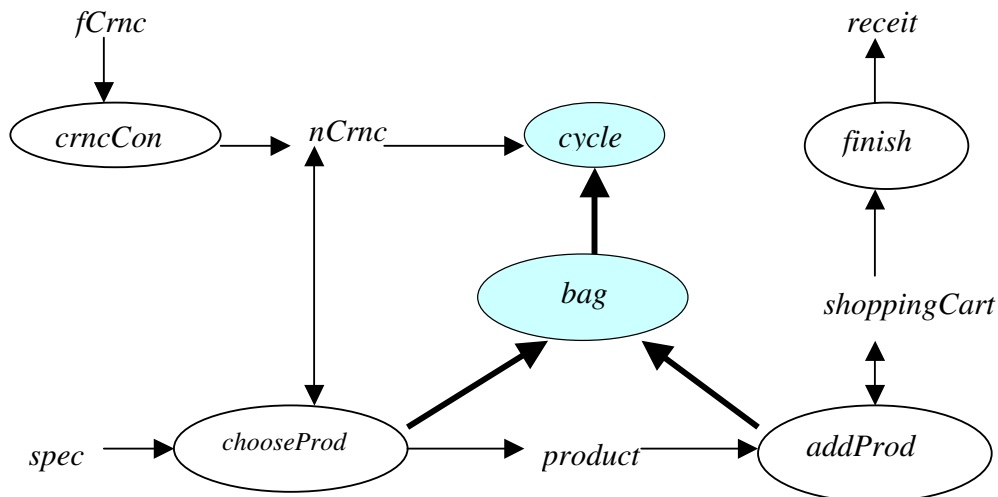


Figure 2. Workflow with data dependencies.

Looking at the arrows in Fig. 2 we can distinguish the higher-order connections between tasks and first order connections between data and tasks. Higher order nodes are *cycle* and *bag*, i.e. the control nodes. The *cycle* is repeating the execution of the whole *bag* as long as there is enough money shown by the value of *nCrnc*, and it is just a while loop.

There are several equivalent formalisms for representing higher order dependencies that cover both data dependencies and control: higher-order functional constraints (HOFCN) [12], higher-order dataflow schemas [6] and a fragment of intuitionistic logic [5]. The schema in Fig. 2 already is an example of a HOFCN. The task and control nodes here can be interpreted as functional constraints, and data nodes as variables. Some constraints (*cycle* and *bag*) bind other functional constraints – they are of higher order. Still, we are going to use logic for defining the semantics of workflow. Representation in logic has the advantage that it allows to reason in a very precise way about the workflow. In particular, it allows us to check whether a given goal is reachable on a workflow model, and if it is, then to synthesize an algorithm for reaching the goal.

3. Representing functionality of tasks in propositional logic

As long as the functionality of tasks of a workflow is expressed only on the level of computability of outputs, and pre- and postconditions are simple propositions as this is the case in web services, one can use intuitionistic propositional logic for representing the semantics of workflow. For example, the functionality of tasks *chooseProduct* and *addProduct* is representable by the following implications where the data names have the role of propositions:

$$\begin{aligned} spec \wedge nCrnc &\supset product \wedge nCrnc \\ product \wedge shoppingCart &\supset shoppingCart \end{aligned}$$

The first tells us that having *spec* and *nCrnc* one can obtain *product* and *nCrnc* (actually, a new value of *nCrnc*). The second tells us that having *product* and *shoppingCart* one can obtain a new value of *shoppingCart*.

According to standard semantics of intuitionistic logic the meaning of a proposition is not a truth-value. It is an object of a proper type [4], and in our case it will always be the data item that is the respective input or output of the task. According to semantics of intuitionistic logic, the implications have a computational meaning, or in other words – their realizations must be functions. In our case they are precisely the tasks *chooseProduct* and *addProduct* that can be shown in a conventional way as the realizations of the formulas written after colon:

$$\begin{aligned} spec \wedge nCrnc &\supset product \wedge nCrnc : chooseProduct \\ product \wedge shoppingCart &\supset shoppingCart : addProduct \end{aligned}$$

To be precise, we should also show the realizations of propositions as variables and realizations of formulas as lambda expressions, for instance, for the second implication it will be as follows:

$$Product : x \wedge shoppingCart : y \supset shoppingCart : \lambda x, y. addProduct$$

However, in the present paper we simplify this formalism and use only names of tasks without losing the correctness. Another option for representing the realizations could be π -calculus [2] as it has been done, for example in [10]. The fragment of logic used here is already sufficient for representing dataflow and “synthesizing” web services as sequences of tasks considering their data dependencies. We postpone the discussion of derivation of sequences of tasks until we have introduced also logic of control structures.

4. Intuitionistic propositional logic represents control constructs

Let us look now at the control of processes presented by workflow, i.e. at the higher-order tasks. There are two higher-order tasks in Fig.1 that are the control structures *cycle* and *sequence*. In order to use logic for control of execution of tasks we have to introduce pre- and postconditions explicitly for each task. They can be connected with tasks by the arrows in the same way as inputs and outputs. Fig. 3 shows the workflow graph with tasks and pre- and postconditions shown explicitly. Instead of an arrow expressing the order of execution of two conventional tasks we have now a propositional variable *ci* that is a postcondition for one and a precondition for another task. Instead of an arrow binding a control structure with a task we have two propositional variables denoting the pre- and postcondition of the task node that is the input of the higher-order node. We have also added two logical variables that denote the conditions of beginning and ending the whole business process.

The logic behind this workflow is expressed by six formulas, one formula for every task or control structure in the workflow. Let us look now at the control structure *cycle*. It has a precondition $c1$ and a postcondition $c2$. Besides that, it produces a precondition $c8$ for the sequence and requires a postcondition $c9$ of the sequence. This is expressed in logic as follows: $c1 \wedge (c8 \supset c9) \supset c2 : cycle$. Indeed, the implication $c8 \supset c9$ represents the functional input realized by *sequence*, hence it must be on the input side of the main implication for *cycle*. It is important to understand that the implication $c8 \supset c9$ represents an input of the control structure *cycle*, but its value is not data! It is the body of the loop performed by the control structure *cycle*. Here is the higher order hidden now. This body must be synthesized from other available tasks, and we call such implication on the left side of another implication a subtask.

Another control node *sequence* has the logical representation $c8 \wedge (c4 \supset c5) \wedge (c6 \supset c7) \supset c9 : sequence$, it has two subtasks: $c4 \supset c5$ and $c6 \supset c7$ whose values have to be synthesized as well. All other nodes are described by simple implications:

$start \supset c1 : crncConv$; $c4 \supset c3 \wedge c5 : chooseProd$; $c3 \wedge c6 \supset c7 : addProd$; $c2 \supset result : finish$.

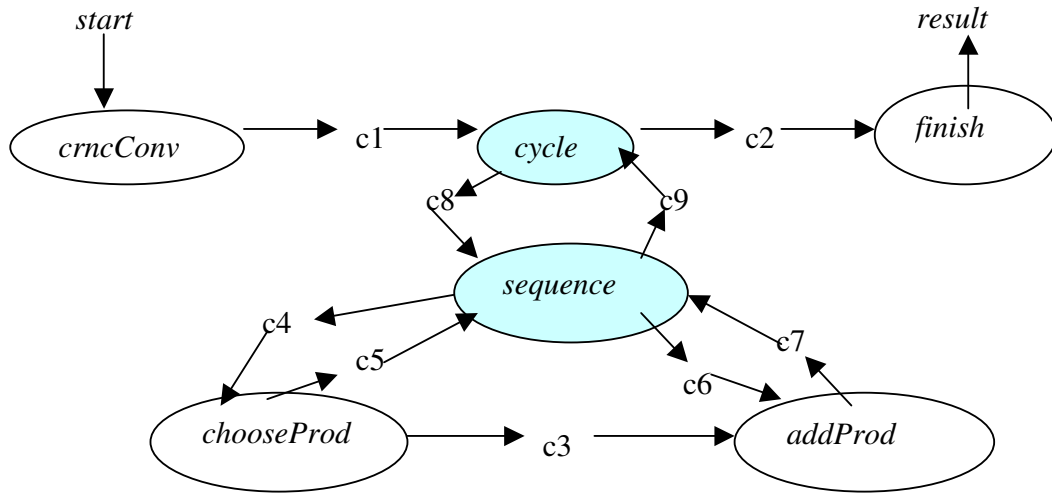


Figure 3. Tasks with pre- and postconditions

Looking more closely at Figure 3, we can see that the information about order of execution of tasks *chooseProd* and *addProd* is represented twice. Indeed, already the logical variable $c3$ shows that these tasks have to be executed sequentially. The same is expressed by the control construct *sequence*. We can simplify the workflow graph as follows:

1. drop the control construct *sequence*
2. drop the pre- and postconditions of sequence except $c8$ and $c9$ that show that some tasks have to be repeatedly executed
3. give to $c8$ and $c9$ the role of pre- and postcondition of the body of *chooseProd* and *addProd* respectively by binding them with these tasks.

The result will be a graph shown in Fig. 4. This workflow graph is simpler than the one in Fig. 3 and still expresses the exact order of execution of tasks. Now $c8$ and $c9$ have roles both as pre- and postconditions of individual tasks and of the whole loop. We denote their connection with *cycle* by dashed lines showing that they represent a subtask.

One can do still better by using data dependencies instead of pre- and postconditions for describing the execution order where possible. Let us look at a workflow with detailed data dependencies shown in Fig. 5. The data items *nCrnc* and *shoppingCart* are represented there as three different instances each. The amount of money *nCrnc* is the data given as an input to *cycle*, the amount *nCrnc'* is passed to *chooseProduct* task at every repetition of the cycle body, and *nCrnc''* is the decreased value of the money after choosing a new product. One can generalize that and say that *nCrnc* is the initial value, *nCrnc'* is the current value and *nCrnc''* is the next value of the data. Also the *shoppingCart* is split in the similar way. The formula for *cycle* is now as follows:

$nCrnc \wedge (nCrnc' \wedge shoppingCart' \supset nCrnc'' \wedge shoppingCart'') \supset shoppingCart : cycle$

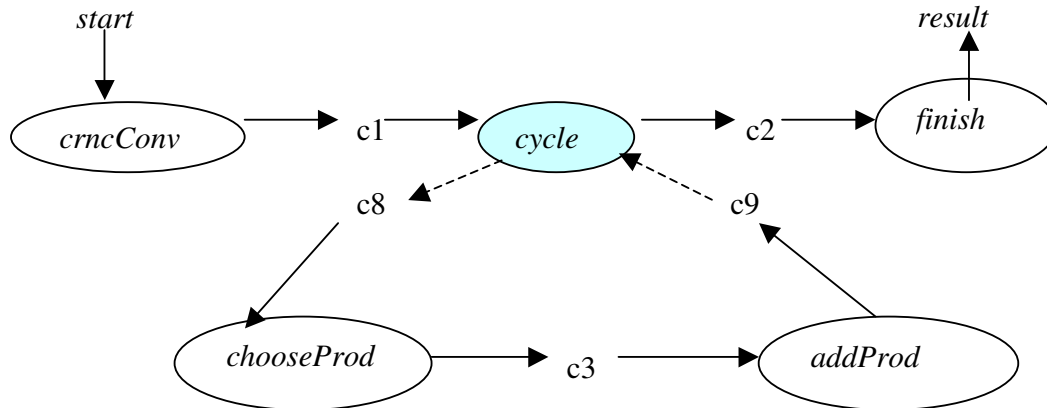


Figure 4. Workflow with a subtask

This formula reads “given $nCrnc$ and a function for computing $nCrnc$ ” and $shoppingCart$ ” from given $nCrnc$ ’ and $shoppingCart$ ’ the $cycle$ can compute $shoppingCart$. This is the exact meaning of this formula in intuitionistic logic. Now comes a surprise again – we do not need the control node bag any more, because the detailed dataflow contains the needed information for synthesizing the body of loop, and the formula for $cycle$ says that the body has to be synthesized.

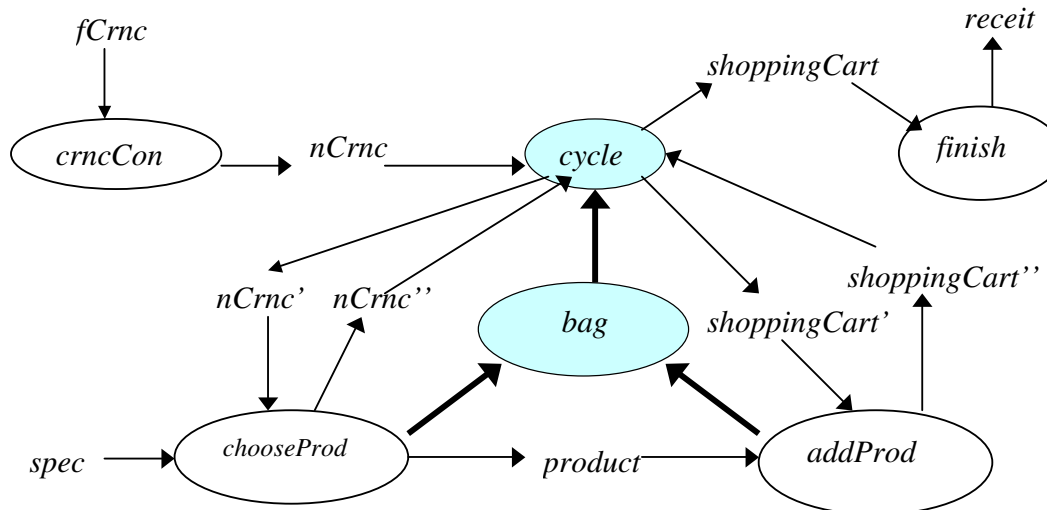


Figure 5. Workflow with detailed data dependencies

4. Automatic composition of processes

The detailed dataflow is sufficient for composing the correct process of computing the $receipt$ from given $fCrnc$ and $spec$. This can be demonstrated by deriving in intuitionistic logic the goal of computations expressed by the implication

$$fCrnc \wedge spec \supset receipt .$$

This can be derived from the formulas that represent four tasks $crncCon$, $chooseProd$, $addProd$, $finish$ and the control structure $cycle$. This derivation is not very short, and the search may be quite time consuming in general, because it is known that the problem of proof search in intuitionistic propositional logic is decidable, but it is P-space complete. However, a good strategy exists for proof search in the case of formulas restricted to implications as we have them for representing tasks and control structures. This strategy is the

basis of structural synthesis of programs [4], and it has been implemented in several tools, the most recent tool is COCOVILA [3]. This strategy combines forward search over tasks and backward search over control structures. The trick is that we use admissible rules for derivation that work better than conventional inference rules of intuitionistic logic in our case. (An admissible rule corresponds to a fragment of derivation in standard logic, hence derivation with admissible rules is shorter than a standard derivation.) There are two admissible rules: one for tasks and another for control structures. Application of a rule corresponds to performing a task or applying a control structure. The rules are as follows (we present here for brevity the rule SS1 with only one subtask, but this is not a principal restriction on the logic):

$$\frac{(A \supset B) \wedge X \supset Z : f \quad A \wedge W \supset B : g}{X \wedge W \supset Z : f;g} \quad (\text{SS1})$$

$$\frac{A \supset B \wedge C : f \quad B \wedge D \supset G \wedge E : g}{A \wedge D \supset C \wedge G : f(g)} \quad (\text{SS2})$$

The letters A, B, C, D, E, G, W, Z are metavariables that denote propositions (data names in our case) or their conjunctions. (We have omitted structural rules here that are applicable and needed as well.) The derivation rules show us also how realizations of consequents are built from realizations of premises of the rules: the rule SS1 composes a sequence of tasks $f;g$, and the rule SS2 takes a body of loop g and gives it as an argument to the realization f of the control structure. (This body must be synthesized in advance as realization of the formula $B \wedge D \supset G \wedge E$.)

Now we can build the derivation of the goal $fCrnc \wedge spec \wedge receipt$ specified in Fig. 6. This demonstrates how the logic is used for finding the order of application of tasks together with control over their execution. The composition of tasks is shown in small Courier font after colon at consequents of each derivation step. The complete description of execution order appears at the last step. Pay attention to the procedural parameter `chooseProd;addProd` of the control structure `cycle` – this is the body of the loop synthesized automatically from two tasks without using the bag node of the graph. As the derivation steps correspond to the nodes of the workflow graph, the proof-search is easy.

$$\frac{spec \wedge nCrnc' \supset nCrnc'' \wedge product \quad shoppingCart' \wedge product \supset shoppingCart''}{spec \wedge nCrnc' \wedge shoppingCart' \supset nCrnc'' \wedge shoppingCart''} : \text{chooseProd;addProd} \quad (\text{SS2})$$

$$\frac{nCrnc \wedge (nCrnc' \wedge shoppingCart' \supset nCrnc'' \wedge shoppingCart'') \supset shoppingCart}{spec \wedge nCrnc \supset shoppingCart} : \text{cycle}(\text{chooseProd;addProd}) \quad (\text{SS1})$$

$$\frac{fCrnc \supset nCrnc}{fCrnc \wedge spec \supset shoppingCart} : \text{crncCon;cycle}(\text{chooseProd;addProd}) \quad (\text{SS2})$$

$$\frac{shoppingCart \supset receipt}{fCrnc \wedge spec \supset receipt} : \text{crncCon;cycle}(\text{chooseProd;addProd});\text{finish} \quad (\text{SS2})$$

Figure 6. Derivation of the goal $fCrnc \wedge spec \supset receipt$

The logical semantics of tasks and control structures allows us to use a collection of tasks as a specification. We can state a goal and try to construct a composite task (a service) that achieves the goal using the given specification. This allows us to look at the collection of available services in a new way. We can speak about a set of goals achievable on the given collection of available services. Let us assume now that in our example the nodes `chooseProd` and `addProd` are collections of more primitive tasks, and the following

goals are solvable: $spec \supset product$ and $product \supset shoppingCart$ on $chooseProd$ and $addProd$ respectively. Then the goal $spec \supset addProd$ is also solvable on the whole set of tasks of the example.

5. Conclusions

We have shown how to represent higher-order workflow in intuitionistic propositional logic, first, by adding pre- and postconditions to every task and control structure, and second, by using only data dependencies. We presented admissible inference rules that enable one to compose complex tasks efficiently.

A fundamental unsolved problem is how the semantic web, including web services, behaves as a knowledge-based self-organizing system. Its formal description and analysis are prerequisites of understanding this behavior. We have presented a formal description of workflow of services and have investigated a concrete practical problem of automatic composition of services. We hope that this can provide also some understanding of the behavior of the semantic web.

We did not tackle the problem of consumable resources here. Multiplicative conjunction of linear logic can be added to the logical language, and this will provide the means for expressing the usage of consumable resources and nonmonotonic conditions, see [9]. We hope that this will not interfere with the logic for handling subtasks of control structures, hence one will be able to construct good admissible rules for this more general case as well.

References

1. van der Aalst, W.M.P. Cerami, E. Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL. O'Reilly (2002).
2. Bellin, G., Scott, P. On the pi-Calculus and Linear Logic. *Theoretical Computer Science*. 135(1) (1994) p. 11 – 65.
3. Grigorenko, P., Saabas, A., Tyugu, E. Visual Tool for Generative Programming. *Proc. of the Joint 10th European Software Engineering Conference (ESEC) and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13)*. ACM Press (2005) p. 249 – 252.
4. Matskin, M., Tyugu, E. Strategies of Structural Synthesis of Programs and Its Extensions. *Computing and Informatics*. v.20 (2001) 1 -25.
5. Mints, G. A Short Introduction to Intuitionistic Logic. Series: [University Series in Mathematics](#). Springer (2001).
6. Mints, G., Tyugu, E. Justification of structural synthesis of programs. *Science of Computer n Programming*, 2(3) (1982) 215—240.
7. Navathe, S., Wakayama, T. Three Good reasons for Using aPetri-Net-asedWorkflowManagement System. Proc. of the International Working Conference on Information and Process Itegration in Enterprises, Cambridge, Massachusetts (1996) p. 179 – 201.
8. OASIS Web Services Business Process Execution Language (WSBPTEL) TC at http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel
9. Rao, J., Küngas, P., Matskin, M. Application of Linear Logic to Web Service Composition. In Proceedings of the First International Conference on Web Services, ICWS'2003, Las Vegas, CSREA Press (2003) p. 3-9.
10. Rao, J. Semantic Web Service Composition via Logic-Based Program Synthesis. Dr. Ing. Thesis, NTNU, Trondheim (2004).
11. Tyugu, E. Higher-Order Dataflow Schemas. *Theoretical Computer Science*, v.90, (1991) 185-198.
12. Tyugu, E., Uustalu, T. Higher-Order Functional Constraint Networks. *Constraint Programming*. NATO ASI Series F: Computer and Systems Sciences (1993) p. 116 – 139.