Petri Net + Nested Relational Calculus = Dataflow

Jan Hidders[1], Natalia Kwasnikowska[2], Jacek Sroka[3], Jerzy Tyszkiewicz[3], and Jan Van den Bussche[2]

[1] University of Antwerp, Belgium
[2] Hasselt University, Belgium
[3] Warsaw University, Poland

Abstract. In this paper we propose a formal, graphical workflow language for dataflows, i.e., workflows where large amounts of complex data are manipulated and the structure of the manipulated data is reflected in the structure of the workflow. It is a common extension of

– *Petri nets*, which are responsible for the organization of the processing tasks, and
– *Nested relational calculus*, which is a database query language over complex objects, and is responsible for handling collections of data items (in particular, for iteration) and for the typing system.

We demonstrate that dataflows constructed in hierarchical manner, according to a set of refinement rules we propose, are *sound*: initiated with a single token (which may represent a complex scientific data collection) in the input node, terminate with a single token in the output node (which represents the output data collection). In particular they always process all of the input data, leave no "debris data" behind and the output is always eventually computed.

1 Introduction

In this paper we are concerned with the creation of a formal language to define *dataflows*. Dataflows are often met in practice, examples are: *in silico* experiments of bioinformatics, systems processing data collected in physics, astronomy or other sciences. Their common feature is that large amounts of structured data are analyzed by a software system organized into a kind of network through which the data flows and is processed. The network consists of many servers, connected by communication protocols, i.e., HTTP or SOAP. In some cases advanced synchronization of the processing tasks is needed.

There are well-developed formalisms for *workflows* that are based on Petri nets [1]. However, we claim that for dataflows these should be extended with data manipulation aspects to describe workflows that manipulate structured complex values and where the structure of this data is reflected in the structure of the workflow. For this purpose we adopt the data model from the *nested relational calculus* which is a well-known and well-understood formalism in the domain of database theory.

Consequently, in a dataflow, tokens (which are generally assumed to be atomic in workflows) are typed and transport complex data values. Therefore, apart from classical places and transitions, we need transitions which perform operations on such data values. Of course, the operations are those of the nested relational calculus.

This way, the title equation emerges:

$$\text{Petri net} + \text{nested relational calculus} = \text{dataflow.}$$

The resulting language can be given a graphical syntax, thus allowing one to draw rather than write programs in it. This seems very important for a language whose programmers would not be professional computer scientists.

Next, we can give a formal semantics for dataflows. On the one hand, it is crucial, since we believe that formal, and yet executable, descriptions of all computational processes in the sciences should be published along with their domain-specific conclusions. Used for that purpose, dataflows can be precisely analyzed and understood, which is crucial for: (i) debugging by the authors, (ii) effective and objective assessment of their merit by the publication reviewers, and (iii) easy understanding by the readers, once published.

Next, the formal semantics makes it possible to perform formal analysis of the behavior of programs, including (automated) optimization and proving correctness.

We demonstrate the potential of the formal methods by proving the following (presented in an informal manner here).

Theorem. *Dataflows constructed hierarchically, i.e., according to a certain set of refinement rules we propose, are* sound*: initiated with a single token (which may represent a complex data value) in the input node, terminate with a single token in the output node (which represents the output value). In particular they always process all of the input data, leave no "debris data" behind and always terminate without a deadlock.*

We would like to point out that the above theorem is quite general—it applies uniformly to a very wide class of dataflows. Yet not every meaningful dataflow can be constructed hierarchically. However, we believe that the prevailing majority of those met in practice are indeed hierarchical.

Our idea of extending classical Petri nets is not new in general. Colored Petri nets permit tokens to be colored (with finitely many colors), and thus tokens carry some information. In the nets-within-nets paradigm [2] individual tokens have Petri net structure themselves. This way they can represent objects with their own, proper dynamics. Finally, self-modifying nets [3] assume standard tokens, but permit the transitions to consume and produce them in quantities functionally dependent on the occupancies of the places.

To compare, our approach assumes tokens to represent complex data values, which are however static. Only transitions are allowed to perform operations over the tokens' contents. Among them, the unnest/nest pairs act in such a way that the unnest transforms a single token with a set value into a set of tokens, and then the nest transforms the set of tokens back into a single "composite" token.

Also the introduction of complex value manipulation into Petri nets was already proposed in earlier work such as [4]. In this paper a formalism called NR/T-nets is proposed where places represent nested relations in a database schema and transitions represent operations that can be applied on the database. Although somewhat similar, the purpose of this formalism, i.e., representing the database schema and possible operations on it, is very different from the one presented here. For example, the structure of the Petri net in NR/T-nets does not reflect the workflow but only which relations are involved in which operations. Another example is the fact that in Dataflow nets we can easily integrate external functions and tools as special transitions and use them at arbitrary levels of the data structures. The latter is an important feature for describing and managing dataflows as found in scientific settings. Therefore, we claim that, together with other differences, this makes Dataflow nets a better formalism for representing dataflows.

1.1 What Is the Nested Relational Calculus?

The nested relational calculus (NRC) is a query language allowing one to describe functional programs using collection types, e.g. lists, bags, sets, etc. The most important feature of the language is the possibility to iterate over a collection. The only collections used in the following are sets, hence in the description below we ignore other collection types of NRC. NRC contains a set of base types. Moreover, it is allowed to combine these types to form records and sets.

Besides standard constructs enabling manipulation of records and set, NRC contains three constructs **sng**, **map** and **flatten**. For a value v of a certain type, **sng**(v) yields a singleton list containing v. Operation **map**, applied to a function from type τ to σ, yields a function from sets of type τ to sets of type σ.

Finally, the operation **flatten**, given a set of sets of type τ, yields a flattened set of type τ, by taking the union. These three basic operations are powerful enough for specifying functions by structural recursion over collections, cf. [5].

The inspiration for these constructs comes from the category-theoretical notion of a *monad* and the monadic approach to uniformly describe different notions of computation [6].

Under its usual semantics the NRC can already be seen as a dataflow description language but it only describes which computations have to be performed and not in what order, i.e., it is rather weak in expressing control flow. For some dataflows this order can be important because a dataflow can include calls to external functions and services which may have side-effects or are restricted by a certain protocol.

1.2 What are Petri Nets?

A classical Petri net is a bipartite graph with two types of nodes called the *places* and the *transitions*. The nodes are connected by directed edges. Only nodes of different kinds can be connected. Places are represented by circles and transitions by rectangles.

Definition 1 (Petri net). *A Petri net is a triple $\langle P, T, E\rangle$ where:*

- *P is a finite set of places,*
- *T is a finite set of transitions ($P \cap T = \varnothing$),*
- *$E \subseteq (P \times T) \cup (T \times P)$ is a set of edges*

A place p is called an *input place* of a transition t, if there exists an edge from p to t. Place p is called an *output place* of transition t, if there exists an edge from t to p. Given a Petri net $\langle P, T, E\rangle$ we will use the following notations:

$$\begin{aligned} \bullet p &= \{t \mid \langle t,p\rangle \in E\} & p\bullet &= \{t \mid \langle p,t\rangle \in E\} \\ \bullet t &= \{p \mid \langle p,t\rangle \in E\} & t\bullet &= \{p \mid \langle t,p\rangle \in E\} \\ \circ p &= \{\langle t,p\rangle \mid \langle t,p\rangle \in E\} & p\circ &= \{\langle p,t\rangle \mid \langle p,t\rangle \in E\} \\ \circ t &= \{\langle p,t\rangle \mid \langle p,t\rangle \in E\} & t\circ &= \{\langle t,p\rangle \mid \langle t,p\rangle \in E\} \end{aligned}$$

and their generalizations for sets:

$$\begin{aligned} \bullet A &= \bigcup_{x \in A} \bullet x & A\bullet &= \bigcup_{x \in A} x\bullet \\ \circ A &= \bigcup_{x \in A} \circ x & A\circ &= \bigcup_{x \in A} x\circ \end{aligned}$$

where $A \subseteq P \cup T$. Places are stores for *tokens*, which are depicted as black dots inside the places when describing the run a Petri net. Edges define the possible token flow. The semantics of a Petri net is defined as a transition system. A *state* is a distribution of tokens over places. It is often referred to as a *marking* $M \in P \to \mathbb{N}$. The state of a net changes when a transitions *fires*. For a transition t to fire it has to be *enabled*, that is, each of its input places has to contain at least one token. If transition t fires, it *consumes* one token from each of the places in $\bullet t$ and produces one token in each of the places in $t\bullet$.

Petri nets are a well-founded process modeling technique. The interest in them is constantly growing for the last fifteen years. Many theoretical results are available. One of the better studied classes are *workflow nets* used in business process workflow management[1].

Definition 2 (strongly connected). *A Petri net is strongly connected if and only if for every two nodes n_1 and n_2 there is a directed path leading from n_1 to n_2.*

Definition 3 (workflow net). *A Petri net $PN = \langle P, T, E\rangle$ is a workflow net if and only if:*

(i) PN has two special places: source and sink. The source has no input edges, i.e., $\bullet source = \varnothing$, and the sink has no output edges, i.e., $sink\bullet = \varnothing$.
(ii) If we add to PN a transition t^* and two edges $\langle sink, t^*\rangle$, $\langle t^*, source\rangle$, then the resulting Petri net is strongly connected.

1.3 How We Combine NRC and Petri Nets

In this paper we propose a formal, graphical workflow language for data-centric scientific workflows. Since we call the type of workflows that we consider dataflows, we call the proposed language a *Dataflow language*. From NRC we inherit the set of basic operators and the type system. This should make reusing of existing database theory results easy. Dataflows could for example undergo optimization process as database queries do. To deal with the synchronization issues arising from processing of the data by distributed services we will use a Petri-net based formalism which is a clear and simple graphical notation and has an abundance of correctness analysis results. We believe that these techniques can be reused and combined with known results from database theory for verifying the correctness of dataflows which can be described in the proposed language.

2 Dataflow Language

The language we propose is a combination of NRC and Petri nets. We label transitions with labels determining functions or NRC operators, and associate NRC values with the tokens. As is usual with workflows that are described by Petri net we mandate one special input place and one special output place. If there is external communication this is modeled by transitions that correspond to calls to external functions. We use edge labeling to define how values of consumed tokens map onto the parameters of operations represented by transitions. To express conditional behavior we propose edge annotations indicating a condition that the value associated with the token must satisfy, so it can be transferred through the annotated edge. We also introduce two special transitions, unnest and nest, to enable explicit iteration over values of a collection.

A dataflow will be defined by an acyclic workflow net, transition and edge labeling, and edge annotation. The underlying Petri net will be called a *dataflow net*.

Definition 4 (dataflow net). *A workflow net $WFN = \langle P, T, E \rangle$ is a dataflow net if and only if it has no cycles.*

With the presence of the NRC map operation acyclicity seems not to be a strong limitation in real life applications. It has also an advantage as termination is always guaranteed.

2.1 The Type System

The dataflows are strongly typed, which means here that each transition consumes and produces tokens with values of a well determined type. The type of the value of a token is called the *token type*. The type system is similar to that of NRC. We assume a finite but user-extensible set of basic types which might for example be given by:

$$b ::= boolean \mid integer \mid string \mid XML \mid ...$$

where *boolean* contains the boolean values *true* and *false*, *integer* contains all integer numbers, *string* contains all strings and *XML* contains all well-formed XML documents. Although this set can be arbitrarily chosen we will require that it at least contains the *boolean* type. From these basic types we can build complex types as defined by:

$$\tau \ ::= \ b \mid \langle l_1 : \tau_1, ..., l_n : \tau_n \rangle \mid \{\tau\}$$

The type $\langle l_1 : \tau_1, ..., l_n : \tau_n \rangle$, where l_i are distinct labels, is the type of all records having exactly fields $l_1, ..., l_n$ of types $\tau_1, ..., \tau_n$ respectively (records with no fields are also included). Finally $\{\tau\}$ is the type of all finite sets of elements of type τ. For later use we define CT to be the set of all complex types and CV the set of all possible complex values (note that the set of basic types is a subset of the set of complex types).

NRC can be defined on other collection types such as lists or bags. They are also included in existing scientific workflow systems such as in Taverna [7] where for example lists are supported. However, after careful analysis of various use cases in bioinformatics and examples distributed with existing scientific workflow systems we have concluded that sets are sufficient. Moreover, if lists are needed they can be simulated with sets in which the position is indicated by a number, but sets cannot be simulated with lists without introducing some arbitrary order on the elements. This order unnecessarily complicates the definition of the semantics and, as is known from database research, may limit optimization possibilities.

2.2 Edge Labels

Dataflows are not only models used to reason about data-processing experiments but are meant to be executed and produce computation results. Distinguishing transition input edges has to be possible to know how the tokens map onto the operation arguments. This is solved by edge labeling. Only edges leading from places to transitions are labeled. This labeling is determined by the edge naming function $EN : \circ T \rightarrow EL$ (note that $\circ T = P\circ$), where EL is some countably infinite set of edge label names, e.g., all strings over a certain non-empty alphabet.

2.3 Transition Labels

To specify desired operations and functions we label Petri net transitions. The transition labeling is defined by a transition naming function $TN : T \rightarrow TL$, where TL is a set of transition labels. Each transition label determines the number and labeling of input edges as well as the types of tokens that the transition consumes and produces when it fires. For this purpose the input typing and output typing functions are used: $IT : TL \rightarrow CT$ maps each transition label to the input type which must be a tuple type and $OT : TL \rightarrow CT$ maps each transition label to the output type.

2.4 Edge Annotation

To introduce conditional behavior we annotate edges with conditions. If an edge is annotated, then it can only transport tokens satisfying the condition. Conditions are visualized on diagrams in an UML [8] way, i.e., in square brackets. Only edges leading from places to transitions are annotated. There are four possible annotations defined by the edge annotation function:

$$EA : \circ T \rightarrow \{\text{“=}\mathbf{true}\text{”}, \text{“=}\mathbf{false}\text{”}, \text{“=}\varnothing\text{”}, \text{“}\neq\varnothing\text{”}, \varepsilon\}$$

The ε represents annotation absence. The meaning of the rest of the labels is self explanatory. For detailed specification see section 4.

2.5 Place Types

With each place in the dataflow net we associate a specific type that restricts the values that tokens in this place may have. This is represented by a function $PT : P \rightarrow CT$.

2.6 Dataflow

The dataflow net with edge naming, transition naming, edge annotation and place typing functions specifies a dataflow.

Definition 5 (dataflow). *Dataflow is a five-tuple* $\langle DFN, EN, TN, EA, PT\rangle$ *where:*

- $DFN = \langle P, T, E\rangle$ *is a dataflow net,*
- $EN : \circ T \rightarrow EL$ *is an edge naming function,*
- $TN : T \rightarrow TL$ *is a transition naming function,*
- $EA : \circ T \rightarrow \{\text{“=}\mathbf{true}\text{”}, \text{“=}\mathbf{false}\text{”}, \text{“=}\varnothing\text{”}, \text{“}\neq\varnothing\text{”}, \varepsilon\}$ *is an edge annotation function,*
- $PT : P \rightarrow CT$ *is a place type function.*

Since it is not true that in all the dataflows the types of the places are those that are required by the transitions we introduce the notion of *well-typed* dataflows. Informally, a dataflow is well-typed if for each transition (1) the names and types of its input places define a tuple type and it is the input tuple type of the transition, (2) the types of the output places are equal to the output type of the transition and (3) if any of the transitions input edges are annotated then the annotations match the types of the associated input places.

Definition 6 (well typed). *A dataflow* $\langle DFN, EN, TN, EA, PT\rangle$ *is well-typed if and only if for each transition* $t \in T$ *it holds that:*

1. *If* $\circ t = \{(p_1, t), ..., (p_n, t)\}$ *and for all* $1 \leq i \leq n$ *it holds that* $l_i = EN((p_i, t))$ *and* $\tau_i = PT(p_i)$ *then* $IT(TN(t)) = \langle l_1 : \tau_1, ..., l_n : \tau_n\rangle$.
2. *For each* $(t, p) \in t\circ$ *it holds that* $PT(p) = OT(TN(t))$.
3. *For each* $(p, t) \in \circ t$ *it holds that:*

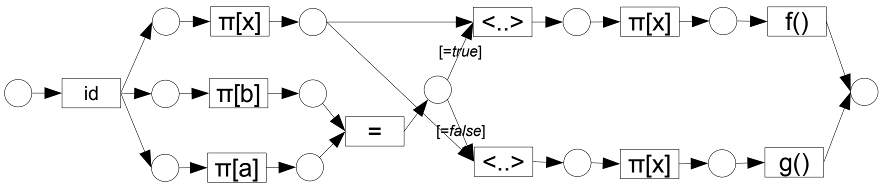

Fig. 1. If-then-else example

- *if* $EA((p,t)) \in \{$"=**true**", "=**false**"$\}$ *then* $PT(p) = $ *boolean, and*
- *if* $EA((p,t)) \in \{$"$=\varnothing$", "$\neq\varnothing$"$\}$ *then* $PT(p)$ *is a set type.*

An example dataflow evaluating an **if** $u = v$ **then** $f(x)$ **else** $g(x)$ expression is shown on Fig. 1. Although the transition labels and a precise execution semantics are defined in the subsequent two sections, the example is self explanatory. First three copies of the input tuple of type $\langle u : b, v : b, x : \tau\rangle$ are made. Then each copy is projected to another field, a and b are compared, and a choice of upper or lower dataflow branch is made on the base of the boolean comparison result. The boolean value is disposed in a projection and depending on the branch that was chosen either $f(x)$ or $g(x)$ is computed.

3 Components

Since it is hard to keep track of new scientific analysis tools and data repositories, the language defines only a *core label subset.* Similarly to NRC the dataflow language can be extended with new *extension transition functions.* Such extension functions will usually represent computations done by external services. Examples from the domain of bioinformatics include: sequence similarity searches with BLAST [9], queries run on the Swiss-Prot [10] protein knowledgebase, or local enactments of the tools from the EMBOSS [11] package.

3.1 Core Transition Labels

The core transition labels are based on the NRC operator set plus two special *unnest* and *nest* labels, as shown in Table 1. A transition label is defined as a combination of the basic symbol (the first column) and a list of parameters which consists of types and edge labels (the second column). The values of the input type function IT and the output type function OT are given by the last two columns. For example, a concrete instance (i.e., with concrete parameter values) of the tuple constructor label would be $tl' = \langle\cdot\cdot\rangle_{a,bool,b,int}$ where the parameters are indicated in subscript and for which the functions IT and OT are defined such that $IT(tl') = OT(tl') = \langle a : bool, b : int\rangle$. Another example would bet $tl'' = \pi[a]_{a,bool,b,int}$ where $IT(tl'') = \langle a : bool, b : int\rangle$ and $OT(tl'') =$ *bool*. The remaining core transition labels are defined in Table 1 in a similar fashion.

Table 1. Core transition labels

Sym.	**Parameters**	**Operation name**	**Input type**	**Output type**
$\varnothing$	l, τ_1, τ_2	empty-set constr.	$\langle l : \tau_1 \rangle$	$\{\tau_2\}$
$\{\cdot\}$	l, τ	singleton-set constr.	$\langle l : \tau \rangle$	$\{\tau\}$
$\cup$	l_1, l_2, τ	set union	$\langle l_1 : \{\tau\}, l_2 : \{\tau\} \rangle$	$\{\tau\}$
φ	l, τ	flatten	$\langle l : \{\{\tau\}\} \rangle$	$\{\tau\}$
$\times$	l_1, τ_1, l_2, τ_2	Cartesian product	$\langle l_1 : \{\tau_1\}, l_2 : \{\tau_2\} \rangle$	$\{\langle l_1 : \tau_1, l_2 : \tau_2 \rangle\}$
$=$	l_1, l_2, b	atomic-value equal.	$\langle l_1 : b, l_2 : b \rangle$	*boolean*
$\langle\rangle$	l, τ	empty tuple constr.	$\langle l : \tau \rangle$	$\langle\rangle$
$\langle\cdot\cdot\rangle$	$l_1, \tau_1, ..., l_n, \tau_n$	tuple constr.	$\langle l_1 : \tau_1, ..., l_n : \tau_n \rangle$	$\langle l_1 : \tau_1, ..., l_n : \tau_n \rangle$
$\pi[l_i]$	$l, \langle l_1 : \tau_1, ..., l_n : \tau_n \rangle$	field projection	$\langle l : \langle l_1 : \tau_1, ..., l_n : \tau_n \rangle \rangle$	τ_i
id	l, τ	identity	$\langle l : \tau \rangle$	τ
$*$	l, τ	unnest	$\langle l : \{\tau\} \rangle$	τ
$*^{-1}$	l, τ	nest	$\langle l : \tau \rangle$	$\{\tau\}$

3.2 Extension Transition Labels

Next to the set of core transition labels the set of transition labels TL also consists of user-defined transition labels. As for all transition labels the functions IT and OT must be defined for each of them. Moreover, for every user-defined transition label tl we will assume that there exists an associated function $\Phi_{tl} : IT(tl) \rightarrow OT(tl)$ which represents a possibly non-deterministic computational function that is performed when the transition fires.

To give a concrete example a bioinformatician may define a $getSWPrByAC$, for which $IT(getSWPrByAC) = \langle ac : string \rangle$ and $OT(getSWPrByAC) = XML$. The $\Phi_{getSWPrByAC}$ function would represent a call to a Swiss-Prot knowledgebase and return a XML formated entry for a given primary accession number.

4 Transition System Semantics

Let the $\langle DFN, EN, TN, EA, PT \rangle$ be a well-typed dataflow (if not stated otherwise this will be assumed in the rest of the paper). Its semantics is given as a transition system (see subsection 4.2). Each place contains zero or more *tokens*, which represent data values. Formally a token is a pair $k = \langle v, h \rangle$, where $v \in CV$ is the transported value and $h \in H$ is this value's *unnesting history.* This unnesting history is defined in the next subsection (see 4.1). The set of all possible tokens is then $K = CV \times H$. By the type of a token we mean the type of its value, i.e., $\langle v, h \rangle : \tau$ if and only if $v : \tau$.

The state of a dataflow, also called *marking*, is the distribution of tokens over places $M \in (P \times K) \rightarrow \mathbb{N} \cup \{0\}$ where $M(p, k) = n$ means that place p contains n times the token k. We only consider as states distributions for which token types match types of places they are in, i.e., for all places $p \in P$ and tokens $k \in K$ such that $M(p, k) > 0$ it holds that $k : PT(p)$. Transitions are the active components in a dataflow: they can change the state by *firing*, that is consuming tokens from each of their input places and producing tokens in each of their output places. In distinction to workflow nets, for some transitions (nests and unnests) more than one token per input place can be consumed and an arbitrary number of tokens per output place can be produced. A transition that can fire in a given state is called *enabled*. The types, numbers and unnesting history conditions of tokens in input places for a given transition to be enabled are determined by its transition label.

We adopt the following Petri net notations:

– $M_1 \xrightarrow{t} M_2$: the transition t is enabled in state M_1 and firing t in M_1 results in state M_2
– $M_1 \rightarrow M_2$: there is a transition t such that $M_1 \xrightarrow{t} M_2$
– $M_1 \xrightarrow{\theta} M_n$: the firing sequence $\theta = t_1 t_2 ... t_{n-1}$ leads from state M_1 to state M_n, i.e., $\exists_{M_2,M_3,...,M_{n-1}} M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} M_3 \xrightarrow{t_3} ... \xrightarrow{t_{n-1}} M_n$
– $M_1 \xrightarrow{*} M_n$: $M_1 = M_n$ or there is a firing sequence $\theta = t_1 t_2 ... t_{n-1}$ such that $M_1 \xrightarrow{\theta} M_n$

A state M_n is called *reachable* from M_1 if and only if $M_1 \xrightarrow{*} M_n$.

Although the semantics of a dataflow is presented as a transition system, as in classical Petri nets, two enabled transitions may fire concurrently, if there is enough input tokens.

4.1 Token Unnesting History

Every time it fires, an unnest transition (see 4.2) consumes one token with a set value and produces a token for each element in this set. The information about the unnested set and the particular element of that set for which a given token was created is stored in that token's unnesting history. This is illustrated in Fig. 2 where in (a) we see in the first place a single token with value $\{1, 2, 3\}$ and an empty history (). When the unnest transition fires it produces a token for each element as shown in (b). The history is then extended with a pair that contains (1) the set that was unnested and (2) the element for which this particular token was produced. As shown in (c) normal transitions will produce tokens with histories identical to that of the consumed input tokens. Once all the tokens that belong to the same unnesting group have arrived in the input place of the nest transition as is shown in (d), which can be verified by looking at their histories, then the nest transition can fire and combine them into a single token as is shown in (e). Note that where the unnest transition adds a pair to the history, the nest transition removes a pair from the history. Since sets can be unnested and nested several times, the history is a *sequence* of pairs where each pair contains the unnesting information of one unnesting step. Therefore we formally define the *set of all histories* H as the set of all sequences of pairs $\langle s, x\rangle$, where $s \in CV$ is a set and $x \in s$.

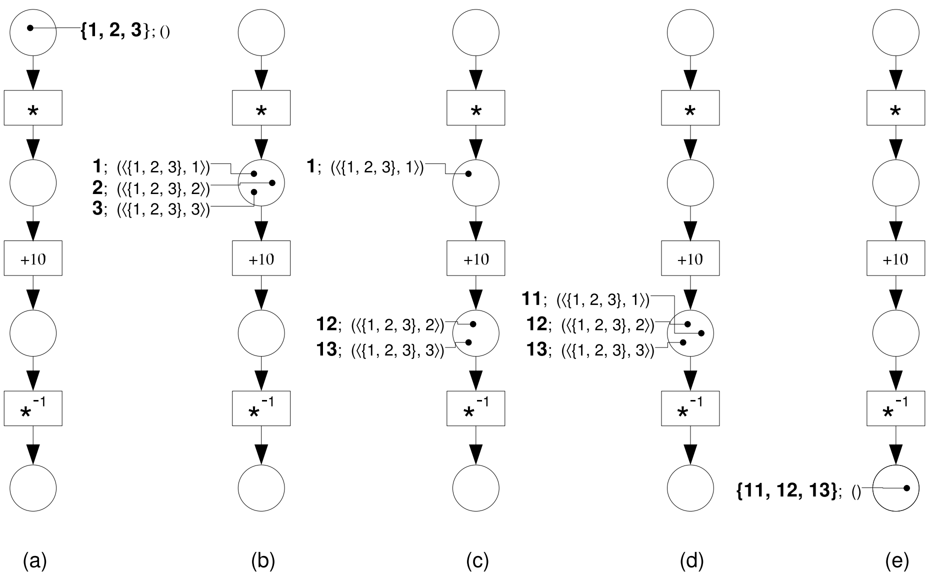

Fig. 2. An illustration of the unnesting history of tokens

4.2 Semantics of Transitions

The following shortcut will be used, since tokens can only flow along a condition-annotated edge if the value of the token satisfies the condition:

$$\begin{aligned}\langle v, h\rangle \curvearrowright e \overset{\text{def}}{=} &(EA(e) = \varepsilon \Rightarrow \mathbf{true}) \wedge \\ &(EA(e) = \text{“}{=}\mathbf{true}\text{”} \Rightarrow v = \mathbf{true}) \wedge \\ &(EA(e) = \text{“}{=}\mathbf{false}\text{”} \Rightarrow v = \mathbf{false}) \wedge \\ &(EA(e) = \text{“}{=}\varnothing\text{”} \Rightarrow v = \varnothing) \wedge \\ &(EA(e) = \text{“}{\neq}\varnothing\text{”} \Rightarrow v \neq \varnothing)\end{aligned}$$

A formal definition for unnest, nest and extension transition labels will be given. The nest and unnest are special since only transitions labeled in this way change the token's unnesting history. Moreover nest transitions can consume more than one token per input place and unnest transitions can produce more than one token per output place. Transitions labeled by other labels behave as in classical Petri nets except that each time they fire all consumed tokens must have identical histories. The semantics of the rest of the core transition labels fully agrees with the intuitions given by their names. In particular, they consume and produce exactly one token when firing, do not change the unnesting history and the formulas are analogous to those given for extension transition labels.

Unnest. For an unnest transition $t \in T$ it holds that $M_1 \xrightarrow{t} M_2$ if and only if there exists a token $\langle v, h\rangle \in K$ such that:

1. for all places $p \in \bullet t$ it holds that:
   (a) $\langle v, h \rangle \curvearrowright \langle p, t \rangle$,
   (b) $M_2(p, \langle v, h \rangle) = M_1(p, \langle v, h \rangle) - 1$ and
   (c) $M_2(p, \langle v', h' \rangle) = M_1(p, \langle v', h' \rangle)$ for all tokens $\langle v', h' \rangle \neq \langle v, h \rangle$
2. for all places $p \in t\bullet$ it holds that:
   (a) $M_2(p, \langle x, h \oplus \langle v, x \rangle \rangle) = M_1(x, \langle v, h \oplus \langle v, x \rangle \rangle) + 1$ for every $x \in v$ and
   (b) $M_2(p, \langle v', h' \rangle) = M_1(p, \langle v', h' \rangle)$ if $\langle v', h' \rangle \neq \langle x, h \oplus \langle v, x \rangle \rangle$ for all $x \in v$
3. for all places $p \notin \bullet t \cup t\bullet$ it holds that $M_2(p, \langle v', h' \rangle) = M_1(p, \langle v', h' \rangle)$ for all tokens $\langle v', h' \rangle \in K$

where $(a_1, a_2, ..., a_n) \oplus a_{n+1} := (a_1, a_2, ..., a_n, a_{n+1})$.

Nest. For a nest transition $t \in T$ it holds that $M_1 \xrightarrow{t} M_2$ if and only if there exists a history h_S, a set $s = \{x_1, ..., x_n\} \in CV$ and a set of tokens $S = \{\langle v_1, h_S \oplus \langle s, x_1 \rangle \rangle, ..., \langle v_n, h_S \oplus \langle s, x_n \rangle \rangle\} \subseteq K$ such that

1. for all places $p \in \bullet t$ it holds that:
   (a) $\langle v_i, h_i \rangle \curvearrowright \langle p, t \rangle$ for each $\langle v_i, h_i \rangle \in S$,
   (b) $M_2(p, \langle v_i, h_i \rangle) = M_1(p, \langle v_i, h_i \rangle) - 1$ for each $\langle v_i, h_i \rangle \in S$,
   (c) $M_2(p, \langle v', h' \rangle) = M_1(p, \langle v', h' \rangle)$ for each $\langle v', h' \rangle \notin S$
2. for all places $p \in t\bullet$ and assuming that $v_S = \{v_1, ..., v_n\}$ it holds that:
   (a) $M_2(p, \langle v_S, h_S \rangle) = M_1(p, \langle v_S, h_S \rangle) + 1$ and
   (b) $M_2(p, \langle v', h' \rangle) = M_1(p, \langle v', h' \rangle)$ for all tokens $\langle v', h' \rangle \neq \langle v_S, h_S \rangle$
3. for all places $p \notin \bullet t \cup t\bullet$ it holds that $M_2(p, \langle v', h' \rangle) = M_1(p, \langle v', h' \rangle)$ for all tokens $\langle v', h' \rangle \in K$

where $(a_1, a_2, ..., a_n) \oplus a_{n+1} := (a_1, a_2, ..., a_n, a_{n+1})$.

Extensions. For an extension transition $t \in T$ that is labeled with an extension transition label it holds that $M_1 \xrightarrow{t} M_2$ if and only if there exists a history h such that with each place $p \in \bullet t$ we can associate a value $v_p \in CV$ such that

1. for all places $p \in \bullet t$ it holds that
   (a) $\langle v_p, h \rangle \curvearrowright \langle p, t \rangle$,
   (b) $M_2(p, \langle v_p, h \rangle) = M_1(p, \langle v_p, h \rangle) - 1$ and
   (c) $M_2(p, \langle v', h' \rangle) = M_1(p, \langle v', h' \rangle)$ for all $\langle v', h' \rangle \neq \langle v_p, h \rangle$
2. for all places $p \in t\bullet$ it holds that if $v = \Phi_{TL(t)}(\langle l_1 : v_1, ..., l_n : v_n \rangle)$ where $\{\langle l_1, v_1 \rangle, ..., \langle l_n, v_n \rangle\} = \{\langle EN(p, t), v_p \rangle \mid p \in \bullet t\}$ then
   (a) $M_2(p, \langle v, h \rangle) = M_1(p, \langle v, h \rangle) + 1$ and
   (b) $M_2(p, \langle v', h' \rangle) = M_1(p, \langle v', h' \rangle)$ if $\langle v', h' \rangle \neq \langle v, h \rangle$
3. for all places $p \notin \bullet t \cup t\bullet$ it holds that $M_2(p, \langle v', h' \rangle) = M_1(p, \langle v', h' \rangle)$ for all tokens $\langle v', h' \rangle \in K$

5 Hierarchical Dataflows

In spite of the fact that we extend workflow Petri nets, existing technical and theoretical results can be easily reused. This is what we intend to demonstrate here.

The dataflow language is developed to model data-centric workflows and in particular scientific data processing experiments. The data to be processed should be placed in the dataflow's source and after the processing ends the result should appear in its sink. A special notation is introduced that distinguishes two state families. The state $input_k$ is an input state with a single token k in the source place and all other places empty, that is $input_k(\langle source, k\rangle) = 1$ and $input_k(\langle p, k'\rangle) = 0$ if $k \neq k'$ or $p \neq source$. Similarly $output_k$ is an output state with single token in the sink place. Starting with one token in the source place and executing the dataflow may not always produce a computation result in the form of a token in the sink place. For some dataflows the computation may halt in a state in which none of the transitions is enabled yet the sink is empty. Even reaching a state in which there are no tokens at all is possible. Examples of such incorrect dataflows are shown on Fig. 3.

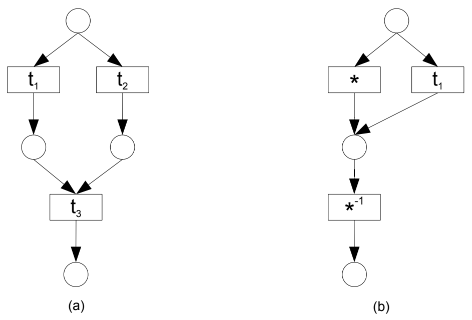

Fig. 3. Incorrect dataflows

For the dataflow (a) the token from the source can be consumed by a transition t1 or t2, but not by both of them at the same time. Transition t3 will not become enabled then, because one of its input places will stay empty. In the (b) case, if t1 gets the source token, the $*^{-1}$ does not become enabled, because only $*$ can produce a token with the required unnesting history. But it may even be not enough when the $*$ transition consumes the source token. If the source token carried an empty set, then in the resulting state all places would be empty.

Similar incorrectness was also studied in context of procedures modeled by classical workflow nets. The correct procedures are called *sound* [1]. The notion of soundness can in a natural way be adapted to dataflows:

Definition 7 (soundness). *A dataflow $\langle DFN, EN, TN, EA, PT\rangle$, with sink : τ and source : θ, is sound if and only if for each token k' : τ there exists token k'' : θ that:*

(i) $\forall_M(input_{k'} \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} output_{k''})$
(ii) $\forall_M(input_{k'} \xrightarrow{*} M \wedge M(out, k'') > 0) \Rightarrow (M = output_{k''})$
(iii) $\forall_{t \in T} \exists_{M,M'} input_{k'} \xrightarrow{*} M \xrightarrow{t} M'$

5.1 Refinement Rules

To avoid designing of unsound dataflows we propose a structured approach. In a *blank dataflow generation step* a pattern dataflow is constructed in a top-down manner, starting from a single place and performing refinements using a given set of rules. Each refinement replaces a place or transition with larger subnet. In the generated *blank dataflow* all transitions are unlabeled, but for each we indicate if it will or will not be labeled with a nest and unnest.

The seven basic refinement rules are shown shown on Fig. 4. Each rule, except the third one, can be applied only if the transformed node has both input and output edges. All the input edges of the transformed node are copied to all the resulting entry nodes, and an analogous rule applies to the output edges. The rules and the aim to make dataflows structured as in structured programming languages were motivated by the work done on workflow nets by Piotr Chrząstowski [12].

Definition 8 (blank dataflow). *A blank dataflow is a tuple* $\langle DFN, NP, EA \rangle$ *where:*

- $DFN = \langle P, T, E \rangle$ *is a dataflow net,*
- $NP : T \rightarrow \{*, *^{-1}, blank\}$ *is a nesting plan function,*
- $EA : E \rightarrow \{$"=**true**", "=**false**", "$=\varnothing$", "$\neq\varnothing$", $\varepsilon\}$ *is an edge annotation function.*

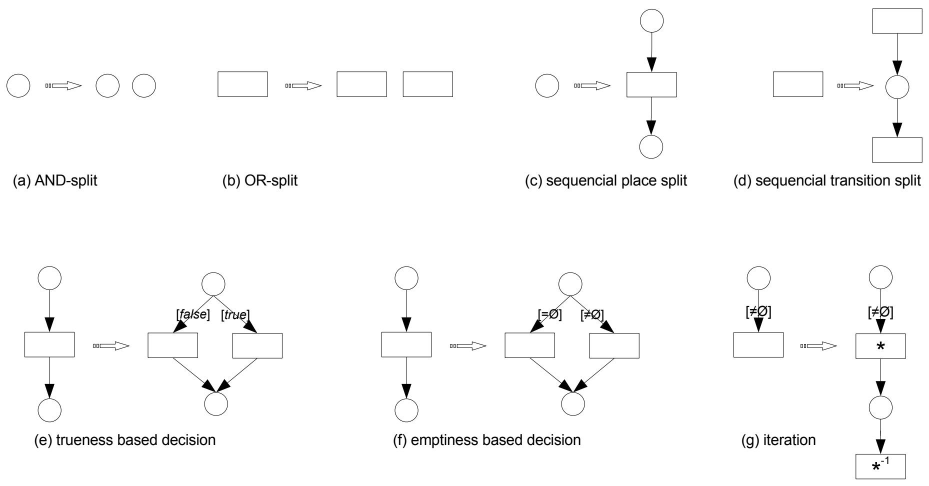
(a) AND-split
(b) OR-split
(c) sequencial place split
(d) sequencial transition split
[false]
[true]
(e) trueness based decision
[=Ø]
[≠Ø]
(f) emptiness based decision
[≠Ø]
[≠Ø]
*
*-1
(g) iteration

Fig. 4. Refinement rules

Definition 9 (blank dataflow generations step). *We start with a blank dataflow* $\langle DFN, NP, EA\rangle$ *consisting of a single place with no transitions and perform the transformations presented on Fig. 4 with the constraint that, except for the sequential place split all the transformations may be applied only if a node has at least one input and at least one output.*

Definition 10 (hierarchical dataflow). *Let* $DF = \langle DFN, EN, TN, EA, PT\rangle$ *be a well-typed dataflow obtained by labeling of all transitions in a blank dataflow net* $BDF = \langle DFN, NP, EA\rangle$ *under following conditions:*

(i) a transition is labeled as nesting if and only if it was planned to, that is: $\exists_{c\in CT} TN(t) = nest_c$ *if and only if* $\forall_{t\in T} NP(t) = *$
(ii) a transition is labeled as unnesting if and only if it was planned to, that is: $\exists_{c\in CT} TN(t) = nest_c$ *if and only if* $\forall_{t\in T} NP(t) = *^{-1}$

The dataflow DF *is hierarchical if and only if the blank dataflow net* BDF *can be constructed in a blank dataflow generation step.*

Theorem 1. *All hierarchical dataflows are sound.*

Proof. (sketch) The proof follows the one given in [12]. It can easily be checked that making any of the refinement rules doesn't jeopardize the possibility of obtaining a sound dataflow by labeling of a blank dataflow. The rest of the proof proceeds by the induction on the number of refinements performed. □

6 A Bioinformatics Dataflow Example

In [13] it was illustrated that NRC is expressive enough to describe real life dataflows in bioinformatics. In this work we combine NRC with Petri nets, using the more convenient Petri net notation for explicitly defining the control flow. In this section we also present a dataflow based on a real bioinformatics example [14]. The corresponding dataflow net is given in Fig. 5. The goal of this dataflow is to find differences in peptide content of two samples of cerebrospinal fluid (a peptide is an amino acid polymer). One sample belongs to a diseased person and the other to a healthy one. A mass spectrometry wet-lab experiment has provided data about observed polymers in each sample. A peptide-identification algorithm was invoked to identify the sequences of those polymers, providing an amino-acid sequence and a confidence score for each identified polymer. The dataflow starts with a tuple containing two sets of data from the identification algorithm one obtained from the "healthy" sample and the other from the "diseased" sample: complex input type $\langle\, \texttt{healthy} : \textit{PepList},\ \texttt{diseased} : \textit{PepList}\,\rangle$ with complex type $\textit{PepList} = \{\,\langle\, \texttt{peptide} : \textbf{String},\ \texttt{score} : \textbf{Number}\,\rangle\,\}$. Each data set contains tuples consisting of an identified peptide, represented by base type **String**, and the associated confidence score, represented by base type **Number**. The dataflow transforms this input into a set of tuples containing the identified peptide, a singleton containing the confidence score from the "healthy" data set or empty set if the identified peptide was absent in the "healthy" data set, and similarly, the confidence score from the "diseased" data set. The complex output type is the following: $\{\,\langle\, \texttt{peptide} : \textbf{String},\ \texttt{healthy} : \{\,\textbf{Number}\,\},\ \texttt{diseased} : \{\,\textbf{Number}\,\}\,\rangle\,\}$.

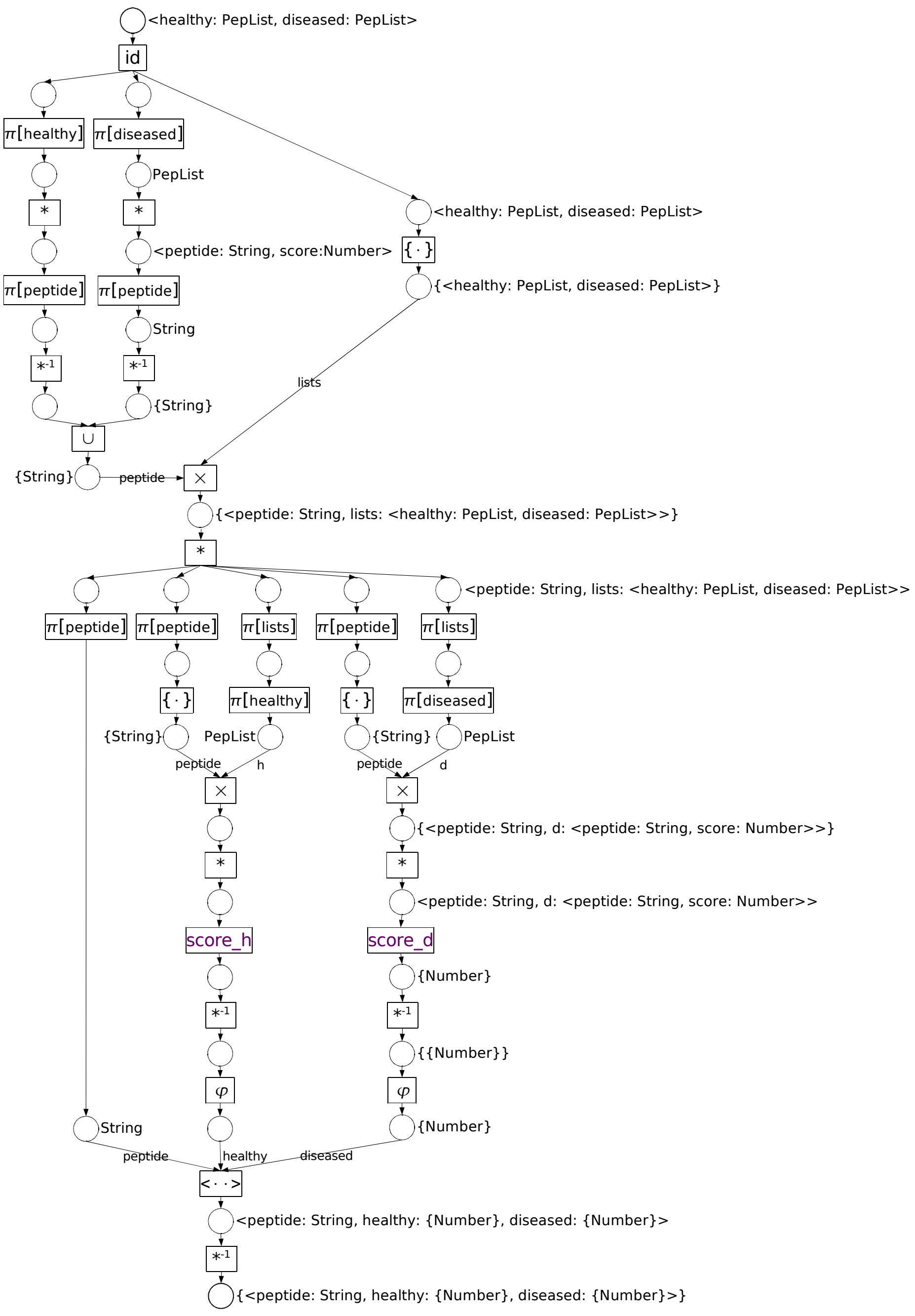
<healthy: PepList, diseased: PepList>
id
π[healthy]
π[diseased]
PepList
*
<healthy: PepList, diseased: PepList>
{·}
<peptide: String, score:Number>
{<healthy: PepList, diseased: PepList>}
π[peptide]
String
lists
*-1
{String}
∪
{String}
peptide
×
{<peptide: String, lists: <healthy: PepList, diseased: PepList>>}
<peptide: String, lists: <healthy: PepList, diseased: PepList>>
π[lists]
π[healthy]
π[diseased]
PepList
h
d
{<peptide: String, d: <peptide: String, score: Number>>}
<peptide: String, d: <peptide: String, score: Number>>
score_h
score_d
{Number}
{{Number}}
φ
healthy
diseased
<···>
<peptide: String, healthy: {Number}, diseased: {Number}>
{<peptide: String, healthy: {Number}, diseased: {Number}>}

Fig. 5. Finding differences in peptide content of two samples

7 Conclusions and Further Research

In this paper we have presented a graphical language for describing dataflows, i.e., workflows where large amounts of complex data are manipulated and the structure of the manipulated data is reflected in the structure of the workflow. In order to be able to describe both the control flow and the data flow the language is based on Petri nets and the nested relational calculus (NRC) and has a formal semantics that is based upon these two formalisms. This ensures that from the large body of existing research on these we can reuse or adapt certain results. This is illustrated by taking a well-known technique for generating sound workflow nets and using it to generate sound dataflow nets. We have shown that the technique that restricts itself to hierarchical dataflows and the technique that goes beyond, can both be readily applied to our formalism.

In future research we intend to compare, investigate and extend this formalism in several ways. Since the dataflow nets tend to become quite large for relatively simple dataflows, we intend to introduce more syntactic sugar. We also want to investigate whether a similar control-flow semantics can be given for the textual NRC and see how the two formalisms compare under these semantics. Since existing systems for data-intensive workflows often lack formal semantics, we will investigate if our formalism can be used to provide these. It is also our intention to add the notions of *provenance* and *history* to the semantics such that these can be queried with a suitable query language such as the NRC. This can be achieved in a straightforward and intuitive way by remembering all tokens that passed through a certain place and defining the provenance as a special binary relation over these tokens. Storing all these tokens makes it not only possible to query the history of a net but also to reuse intermediate results of previous versions of a dataflow. Another subject is querying dataflows where a special language is defined to query dataflow repositories to, for example, find similar dataflows or dataflows that can be reused for the current research problem. Since dataflow nets are essentially labeled graphs it seems likely that a suitable existing graph-based query formalism could be found for this. Finally we will investigate the possibilities of workflow optimization by applying known techniques from NRC research. Since optimization often depends on the changing of the order of certain operations it will then be important to extend the formalism with a notion of "color" for extension transitions that indicates whether their relative order may be changed by the optimizer.

References

1. van der Aalst W.: The application of petri nets to workflow management. The Journal of Circuits, Systems and Computers (1998) 21–66
2. Valk, R.: Object Petri nets: Using the nets-within-nets paradigm. In: Lectures on Concurrency and Petri Nets. (2003) 819–848
3. Valk, R.: Self-modifying nets, a natural extension of Petri nets. In: ICALP. (1978) 464–476

4. Oberweis, A., Sander, P.: Information system behavior specification by high level petri nets. ACM Trans. Inf. Syst. **14** (1996) 380–420
5. Buneman, P., Naqvi, S., Tannen, V., Wong, L.: Principles of programming with complex objects and collection types. Theoretical Computer Science (1995) 3–48
6. Moggi, E.: Notions of computation and monads. Information and Computation (1991) 55–92
7. Oinn, T., Addis, M., Ferris, J., Marvin, D., Greenwood, M., Carver, T., Wipat, A., Li, P.: Taverna: A tool for the composition and enactment of bioinformatics workflows. Bioinformatics (2004)
8. Object Management Group: Unified modeling language resource page. http://www.uml.org/
9. Altschul, S., Gish, W., Miller, W., Myers, E., Lipman, D.: Basic local alignment search tool. J. Mol. Biol. (1990) 403–410
10. Boeckmann, B., Bairoch, A., Apweiler, R., Blatter, M., Estreicher, A., et al.: The swiss-prot protein knowledgebase and its supplement trembl in 2003. Nucleic Acids Research **31** (2003) 365–370
11. Rice, P., Longden, I., Bleasby, A.: Emboss: The european molecular biology open software suite (2000). Trends in Genetics **16** (2000) 276–277
12. Chrząstowski-Wachtel, P., Benatallah, B., Hamadi, R., O'Dell, M., Susanto, A.: A top-down petri net-based approach for dynamic workflow modeling. In: Proceedings of Business Process Management: International Conference, BPM 2003. Volume 2678 of Lecture Notes in Computer Science., Springer (2003) 336–353
13. Gambin, A., Hidders, J., Kwasnikowska, N., Lasota, S., Sroka, J., Tyszkiewicz, J., Van den Bussche, J.: NRC as a formal model for expressing bioinformatics workflows. Poster at ISMB 2005 (2005)
14. Dumont, D., Noben, J., Raus, J., Stinissen, P., Robben, J.: Proteomic analysis of cerebrospinal fluid from multiple sclerosis patients. Proteomics **4** (2004)