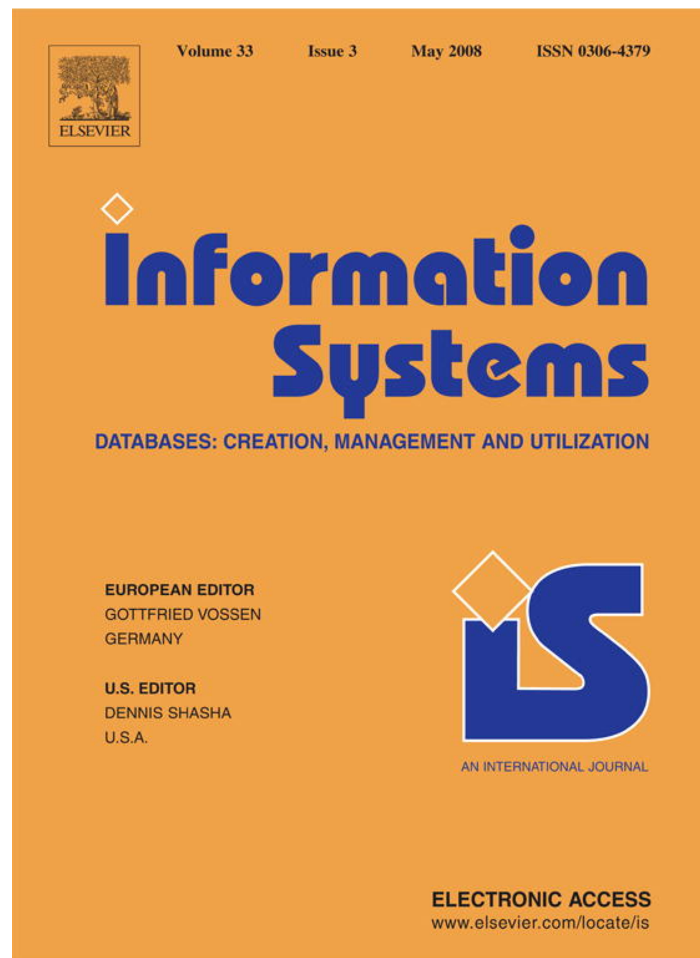
Provided for non-commercial research and education use.
Not for reproduction, distribution or commercial use.



This article was published in an Elsevier journal. The attached copy is furnished to the author for non-commercial research and education use, including for instruction at the author's institution, sharing with colleagues and providing to institution administration.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

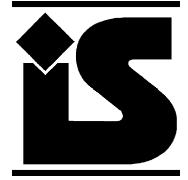
In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

http://www.elsevier.com/copyright

Author's personal copy

Available online at www.sciencedirect.com
ScienceDirect

Information Systems 33 (2008) 261–284



www.elsevier.com/locate/infosys

DFL: A dataflow language based on Petri nets and nested relational calculus ☆

Jan Hidders[a], Natalia Kwasnikowska[c,d], Jacek Sroka[b,*,1], Jerzy Tyszkiewicz[b,1], Jan Van den Bussche[c,d]

[a] *Department of Mathematics and Computer Science, University of Antwerp, Antwerp, Belgium*
[b] *Institute of Informatics, Warsaw University, ul. Banacha 2, 02-097 Warsaw, Poland*
[c] *Theoretical Computer Science Group, Hasselt University, Belgium*
[d] *Transnational University of Limburg, Belgium*

Received 16 November 2006; received in revised form 13 April 2007; accepted 27 September 2007
Recommended by B. Kemme

Abstract

In this paper we propose DFL—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 (1) *Petri nets*, which are responsible for the organization of the processing tasks, and (2) *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 a hierarchical manner, according to a set of refinement rules we propose, are *semi-sound*, i.e., 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 never leave any "debris data" behind and an output is always eventually computed regardless of how the computation proceeds.
© 2007 Elsevier B.V. All rights reserved.

Keywords: DFL; Petri net; Workflow system; Dataflow; Scientific workflow; Nested relational calculus

1. Introduction

In this paper we are concerned with the creation of a formal language to define *dataflows*—DFL (a dataflow language). Dataflows are often met in practice, e.g., *in silico* experiments in bioinformatics and 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

☆ A preliminary version of this paper was presented at the 2005 International Conference on Cooperative Information Systems.

*Corresponding author. Tel.: +48 22 55 44 430; fax: +48 22 55 44 400.

E-mail addresses: jan.hidders@ua.ac.be (J. Hidders), natalia.kwasnikowska@uhasselt.be (N. Kwasnikowska), sroka@mimuw.edu.pl (J. Sroka), jty@mimuw.edu.pl (J. Tyszkiewicz), jan.vandenbussche@uhasselt.be (J. Van den Bussche).

[1] Supported by KBN Grant 4 T11C 042 25.

0306-4379/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2007.09.002

Author's personal copy

is processed. Nodes in the network represent external computations like web-service or local program calls.

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* (NRC), 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 NRC.

The resulting language can be given a graphical syntax, thus allowing one to draw rather than to write programs. This seems very important for a language designed for users that are not professional computer scientists.

Next, we can give a formal semantics for dataflows. This 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 important for: (i) debugging by the authors, (ii) effective and objective assessment of their merit by the reviewers, and (iii) clear understanding by the readers, once published.

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

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

Theorem. *Dataflow constructed hierarchically*, *i.e.*, *according to a certain set of refinement rules we propose*, *is* semi-sound, *i.e.*, *initiated in the input node with a single token representing a scientific data collection*, *terminate with a single token in the output node. In particular it never leaves any "debris data" behind and the output is always eventually computed, regardless of how the computation proceeds.*

We would like to emphasize 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 [2] permit tokens to be colored (with finitely many colors), and thus tokens carry some information. In the nets-within-nets paradigm [3] individual tokens have Petri net structure themselves. This way they can represent objects with their own, proper dynamics. Finally, self-modifying nets [4] 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. The transitions are allowed to perform operations on the tokens' contents. Edges can be annotated with conditions and pass only tokens which values satisfy those conditions. There is also a special unnest/nest annotation. When unnest is applied to an output edge of a transition, the output token with a set value is transformed into a set of tokens, one for each element of the set. When nest is applied to an input edge of a transition, the set of tokens is grouped back into a single "composite" token.

Also the introduction of complex value manipulation into Petri nets was already proposed by others. Oberweis and Sander [5] proposed a formalism called NR/T-nets where places represent nested relations in a database schema and transitions represent operations that can be applied to the database. Although somewhat similar, the purpose of that 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. In our DFL formalism, 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 DFL a better formalism for representing dataflows.

Author's personal copy

An initial version of DFL and the set of refinement rules was presented in [6]. This paper extends that work by giving more elaborate proofs and explaining the semantics of the language in more detail.

1.1. Nested relational calculus

The NRC [7] 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. NRC assumes a set of base types which can be combined to form nested record and collection types. The only collection type we will use are sets.

Besides standard language constructs enabling manipulation of records and sets, NRC contains the three constructs **sng**, **map** and **flatten**. For a value v of a certain type, **sng**(v) yields the singleton set containing v. Operation **map**, applied to a function of type $\tau \to \sigma$, yields a function on sets of type $\{\tau\} \to \{\sigma\}$. 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 [7].

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, such as Web services, which may have side-effects or are restricted by a certain protocol.

1.2. Petri nets

A classical Petri net [8,9] is a bipartite graph with two types of nodes called *places* and *transitions*. The nodes are connected by directed edges. Only nodes of different types 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 ($P \cap T = \emptyset$),
- $E \subseteq (P \times T) \cup (T \times P)$ 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. A place p is called an *output place* of a 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:

$$\bullet p = \{t | \langle t, p\rangle \in E\}, \quad p\bullet = \{t | \langle p, t\rangle \in E\},$$
$$\bullet t = \{p | \langle p, t\rangle \in E\}, \quad t\bullet = \{p | \langle t, p\rangle \in E\},$$
$$\circ p = \{\langle t, p\rangle | \langle t, p\rangle \in E\}, \quad p\circ = \{\langle p, t\rangle | \langle p, t\rangle \in E\},$$
$$\circ t = \{\langle p, t\rangle | \langle p, t\rangle \in E\}, \quad t\circ = \{\langle t, p\rangle | \langle t, p\rangle \in E\}$$

and their generalizations for sets:

$$\bullet A = \bigcup_{x \in A} \bullet x, \quad A\bullet = \bigcup_{x \in A} x\bullet,$$
$$\circ A = \bigcup_{x \in A} \circ x, \quad A\circ = \bigcup_{x \in A} x\circ,$$

where $A \subseteq P \cup T$. Places are stores for *tokens*, which are depicted as black dots inside places when describing the run of 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* $M \in P \to (\mathbb{N} \cup \{0\})$. 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 on 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 15 years. Many theoretical results are available. One of the better studied classes are *workflow nets*, which are used in 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 exists 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: a source and a sink. The *source* has no input edges, i.e., $\circ source = \emptyset$, and the *sink* has no output edges, i.e., $sink\circ = \emptyset$.
(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.

Author's personal copy

1.3. How we combine NRC and Petri nets

In this paper we propose a formal, graphical workflow language for dataflows—data-centric, scientific workflows. We call the proposed language DFL. 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 an 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 DFL.

The fundamental operation in NRC is the map operation *map*. In order to allow a similar kind of iteration in Petri nets we introduce special unnest and nest edges. Unnest edges are outgoing edges of transitions and nest edges are incoming edges. Unnest edges can be used if the function associated with the transition produces a set value. If an outgoing edge is marked as an unnest edge then, if the transition fires, instead of producing in the associated place a single token with the set that is the result of the transition, it will produce a set of tokens, one for each element of the set. Nest edges can be used if the function associated with the transition requires a set value as a parameter. If an incoming edge is marked as a nest edge then, if the transition fires, instead of consuming from the associated place a single token with a set value, it will consume a set of tokens and combine them into a single set that is used as the parameter of the function.

A simple example with a nested iteration is given in Fig. 1. If the dataflow is initiated in the left most place with a token representing a set of sets, it will be processed by the identity transition *id* and unnested. Next, the resulting tokens representing sets that were elements of the input set are unnested themselves by the second pair of identity transition and unnest edge. Finally, function $f(\,)$ is applied to each of the elements of the unnested subsets and the result is twice nested by two subsequent identity transitions with nest edges. To assure that tokens originating from different sets are not intermixed while nesting and that nesting appears only when all the necessary tokens have arrived, each token carries its unnesting history, which is described in Section 4.1.

The unnest and nest edges allow a straightforward representation of the NRC *map* operation in a Petri net formalism and make it possible to reflect the structure of the iteration in the structure of the net, which is desired for data-centric workflows.

2. The DFL language

We define DFL by starting with Petri nets and adding labels to transitions to define the computation done by them. Then we associate NRC values

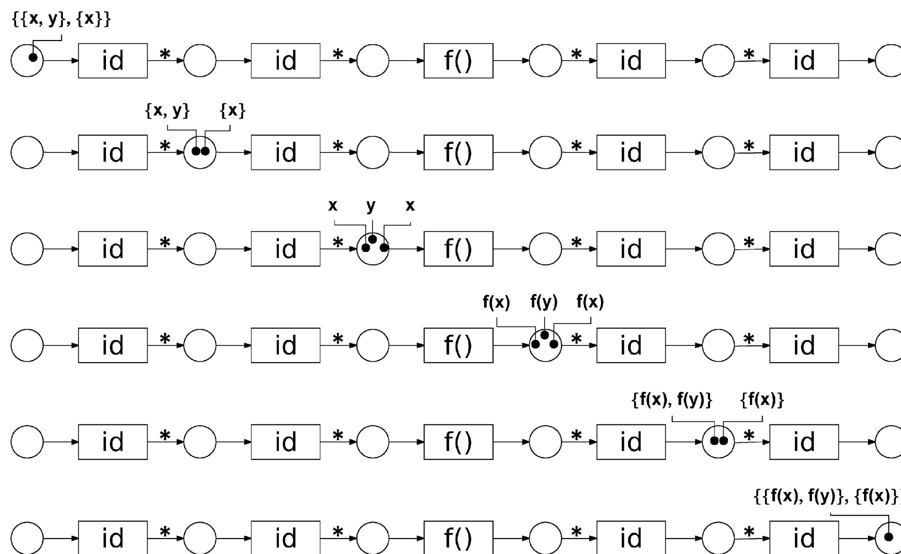


Fig. 1. Nested iteration example.

Author's personal copy

with the tokens to represent the manipulated data. As it is usual with workflows that are described by Petri nets 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 the consumed tokens map onto the parameters of operations represented by transitions. To express conditional behavior we propose edge annotations indicating conditions that the value associated with a token must satisfy, so it can be transferred through the annotated edge. We also introduce a special unnest/nest annotation, to enable explicit iteration over values of a collection.

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

Definition 4 (*Dataflow net*). A $\mathrm{DFN} = \langle P, T, E, \mathit{source}, \mathit{sink}\rangle$ is a dataflow net if and only if:

(i) $\langle P, T, E\rangle$ is a workflow net and is acyclic,
(ii) $\mathit{source} \in P$ is the source place,
(iii) $\mathit{sink} \in P$ is the sink place.

The restriction to acyclic nets is introduced to keep the presentation of the main ideas simple. The formalism can be easily extended such that cycles are allowed. Usually they are used to express iteration over all elements of a list, but for this type of iteration we will introduce alternatives in the form of constructs for unnesting and nesting values. Obviously this does not cover all types of iteration, but we conjecture that it is sufficient for the purpose of scientific dataflows. In addition, an advantages of the restriction is that the termination is always guaranteed, but note that termination does not ensure correct termination, i.e., termination with only one token left which is in the sink and contains the output value.

2.1. The type system

Dataflows are strongly typed, which here means 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*. We will identify a type and the set of objects of that 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

$$b ::= \mathit{boolean} | \mathit{integer} | \mathit{string} | \mathit{XML},$$

where the type *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. Assuming that the non-terminal l denotes the set of field labels, from these basic types we can build complex types as defined by

$$\tau ::= b | \langle l : \tau, \ldots, l : \tau\rangle | \{\tau\}.$$

The type $\langle l_1 : \tau_1, \ldots, l_n : \tau_n\rangle$, where l_i are distinct labels, is the type of all records having exactly fields $l_1, \ldots, l_n$ of types $\tau_1, \ldots, \tau_n$, 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.

NRC can be also defined on other collection types such as lists or bags. Moreover they are included in existing scientific workflow systems, for example Taverna [10] supports lists. However, after a careful analysis of various use cases in bioinformatics and examples distributed with existing scientific workflow systems we have concluded that sets are sufficient.

2.2. Edge naming function

Dataflows are not only models used to reason about data-processing experiments but are meant to be executed and produce computation results. In particular, when a transition has several input edges, we need a way to distinguish those, so as 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 an edge naming function $EN : \circ T \to EL$ (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. The function EN is injective when restricted to incoming edges of a certain transition, i.e., there cannot be two distinct incoming edges with the same edge label for the same transition.

2.3. Transition naming function

To specify the desired operations and functions we also label the transitions. The transition labeling

Author's personal copy

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 possible 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. Note that these two functions are at the global level in the sense that they are the same for every dataflow and therefore not part of the dataflow itself. This is similar to the signatures of system functions which are not part of a specific program. For detailed specification of transition labels see Section 3.

2.4. Edge annotation function

To introduce conditional behavior we annotate edges with conditions. If an edge is annotated with a condition, then it can only transport tokens that satisfy the condition. Conditions are visualized on diagrams in UML [11] fashion, i.e., in square brackets. Only edges leading from places to transitions are annotated with conditions. There are four possible condition annotations: "= true", "= false", "$=\emptyset$", "$\neq\emptyset$". Their meaning is self-explanatory. For detailed specification see Section 4.

There is another annotation "$*$" used to indicate a special unnest/nest branch. On diagrams it is visualized by addition of the symbol "$*$" in the middle of the edge. This annotation can occur on edges leading from transitions to places as well as on edges from places to transitions. When an edge leading from a transition to a place is annotated in such manner, it means that a set value produced by this transition is unnested. That is, instead of inserting a token with a set value into the destination place, a set of tokens representing each element in the set value gets inserted. Such edges will be called *unnest edges*. When an edge leading from a place to a transition is annotated in such manner, it means that in order to fire the destination transition a set of tokens that originated from unnesting of some set value will be used. That is, a set of tokens that originated from unnesting of some set value will be consumed and a set of their values will be an input data for the destination transition. Such edges will be called *nest edges*. The precise semantics and explanation of the mechanism that is used to make sure that all the tokens that originated from unnesting of some set value are already there is described in Section 4.

The annotations are defined by an edge annotation function:

$$EA: (\circ T \rightarrow \{\text{``} = \text{true''}, \text{``} = \text{false''}, \text{``} = \emptyset\text{''}, \text{``}\neq\emptyset\text{''}, \text{``} * \text{''}, \varepsilon\}) \cup (\circ P \rightarrow \{\text{``} * \text{''}, \varepsilon\}),$$

where ε indicates the absence of an annotation.

2.5. Place type function

With each place in a dataflow net we associate a specific type that restricts the allowed values for tokens in that place. This is represented by a place type 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*). A dataflow is a five-tuple $\langle DFN, EN, TN, EA, PT \rangle$ where:

- $DFN = \langle P, T, E, source, sink \rangle$ is a dataflow net,
- $EN: \circ T \rightarrow EL$ is an edge naming function such that for each transition t the partial function $EN|_{\circ t}$ is injective,
- $TN: T \rightarrow TL$ is a transition naming function,
- $EA: (\circ T \rightarrow \{\text{``} = \text{true''}, \text{``} = \text{false''}, \text{``} = \emptyset\text{''}, \text{``}\neq\emptyset\text{''}, \text{``} * \text{''}, \varepsilon\}) \cup (\circ P \rightarrow \{\text{``} * \text{''}, \varepsilon\})$ is an edge annotation function,
- $PT: P \rightarrow CT$ is a place type function.

In order to ensure that the different labelings and annotations in a dataflow are consistent, we introduce the notion of *legality*. Informally, a dataflow is legal, if for each transition t: (1) the input edge labels and the types of their corresponding places, with the nest edges taken into account, define the input type of t; (2) if any of the input edges of t are annotated with conditions, then the annotations are consistent with the types of the associated input places; (3) if an output edge of t is not an unnest edge, then the type of the connected place is equal the output type of t, but if an output edge of t is an unnest edge, then the output type of t is a set type and the type of the connected place is equal to the element type of this set type.

Author's personal copy

Definition 6 (*Legal*). A dataflow $\langle DFN, EN, TN, EA, PT\rangle$ is legal if and only if each transition $t \in T$ satisfies the following:

(1) if $\{\langle p_1, t\rangle, \ldots, \langle p_n, t\rangle\} = \circ t$ and for $1 \leqslant i \leqslant n$ we have

$$l_i = EN(\langle p_i, t\rangle) \quad \text{and}$$

$$\tau_i = \begin{cases} PT(p_i) & \text{if } EA(\langle p_i, t\rangle) \neq \text{``}*\text{''}, \\ \{PT(p_i)\} & \text{if } EA(\langle p_i, t\rangle) = \text{``}*\text{''} \end{cases}$$

then $IT(TN(t)) = \langle l_1 \colon \tau_1, \ldots, l_n \colon \tau_n\rangle$,

(2) for each $\langle p, t\rangle \in \circ t$:
- if $EA(\langle p, t\rangle) \in \{\text{``}=\text{true''}, \text{``}=\text{false''}\}$, then $PT(p) = boolean$, and
- if $EA(\langle p, t\rangle) \in \{\text{``}=\emptyset\text{''}, \text{``}\neq\emptyset\text{''}\}$, then $PT(p)$ is a set type,

(3) for each $\langle t, p\rangle \in t\circ$:
- if $EA(\langle t, p\rangle) \neq \text{``}*\text{''}$, then $OT(TN(t)) = PT(p)$, and
- if $EA(\langle t, p\rangle) = \text{``}*\text{''}$, then $OT(TN(t)) = \{PT(p)\}$.

Henceforth, dataflows will always be assumed to be legal. Legality is an easy syntactic check.

An example dataflow representing an **if** $u = v$ **then** $f(x)$ **else** $g(x)$ expression is shown in Fig. 2. Although the transition labels and a precise execution semantics are defined in the next two sections, the example is self-explanatory. First, three copies of the input tuple of type $\langle u \colon b, v \colon b, x \colon \tau\rangle$ are made. Then, each copy is projected on another field, basic values u and v are compared, and a choice of upper or lower dataflow branch is made on the basis 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. Transition labels

Since it is impossible to gather all scientific analysis tools that one may want to use and data repositories that one may want to query, DFL defines only a *core label subset* that is sufficient to express typical operations on the values from our type system. Similarly to NRC, DFL can be extended with new *extension transition labels*. Such extension labels will usually represent computations done by external services. Examples from the domain of bioinformatics include: sequence similarity searches with BLAST [12], queries to the Swiss-Prot [13] protein knowledgebase, or local enactments of the tools from the EMBOSS [14] package.

3.1. Core transition labels

The core transition labels are based on the NRC operator set and are shown in Table 1. A transition label is defined as a combination of the basic symbol, from the first column, and a list of parameters which consists of types and edge labels, from 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 of the tuple constructor label, i.e., a constructor label with concrete parameter values, 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 $IT(tl') = OT(tl') = \langle a \colon bool, b \colon int\rangle$. Another example would be $tl'' = \pi[b]_{a,b,bool,c,int}$, where $IT(tl'') = \langle a \colon \langle b \colon bool, c \colon int\rangle\rangle$ and $OT(tl'') = bool$.

Moreover, for every transition label tl, there exists an associated function $\Phi_{tl} \colon IT(tl) \to OT(tl)$ which represents a computation that is performed when the transition fires. For the core transition label subset all functions are deterministic and correspond to those given in NRC definition [7].

To keep this presentation simple we will omit edge names and label parameters on diagrams, if it does not introduce ambiguity.

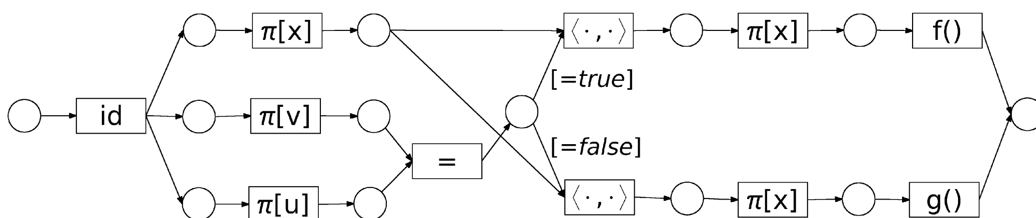


Fig. 2. If–then–else example.

Author's personal copy

Table 1
Core transition labels

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

3.2. Extension transition labels

Next to the set of core transition labels, the set of transition labels TL also contains user-defined transition labels. As for core transition labels the functions IT and OT must be defined for each of them, as well as an associated function $\Phi_{tl}\colon IT(tl) \to OT(tl)$ which can represent a possibly non-deterministic computation that is performed when the transition fires.

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

4. Transition system semantics

The semantics of a dataflow $\langle DFN, EN, TN, EA, PT \rangle$ is defined as a transition system (see Section 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 Section 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 $M : (P \times K) \to (\mathbb{N} \cup \{0\})$ of tokens over places, where $M(p, k) = n$ means that place p contains n copies of the token k. Distributions are legal as markings only if the token types match the types of places their are in, i.e., for all places $p \in P$ and tokens $k \in K$ such that $M(p, k) > 0$ we must have $k : PT(p)$.

Transitions are the active components in a dataflow. They can change the state by *firing*, i.e., consuming tokens from their input places and producing tokens in their output places. In distinction to classical workflow nets, transitions may produce/consume an arbitrary number of tokens in/from a place. This is the case when an edge connecting such a place with the transition is annotated with "$*$", i.e., is an unnest/nest edge. A transition that can fire in a given state will be called *enabled*. Firing of a transition t represents a computation step determined by the function $\Phi_{TN(t)}$ associated with its transition label. Tokens consumed from input places determine the computation's input value with respect to the definitions in Table 1.

4.1. Token unnesting history

Every time a transition with an unnest edge fires, a set of tokens is produced. Each token corresponds to an element of the set value that was produced as a result of a computation carried out by that transition. The history of each of the tokens is extended with a pair that contains the unnested set and an element of that set to which the given token corresponds. The full history is taken into account when it is being determined whether a transition with nest edge can fire, that is if tokens representing all of the elements of the set that is being nested are already there to be consumed. If it is the case, then a set of tokens will be consumed and the set of their values will be used to compute the result.

This is illustrated in Fig. 3. In (a) in the top place we see a single token with value $\{1, 2, 3\}$ and an

Author's personal copy

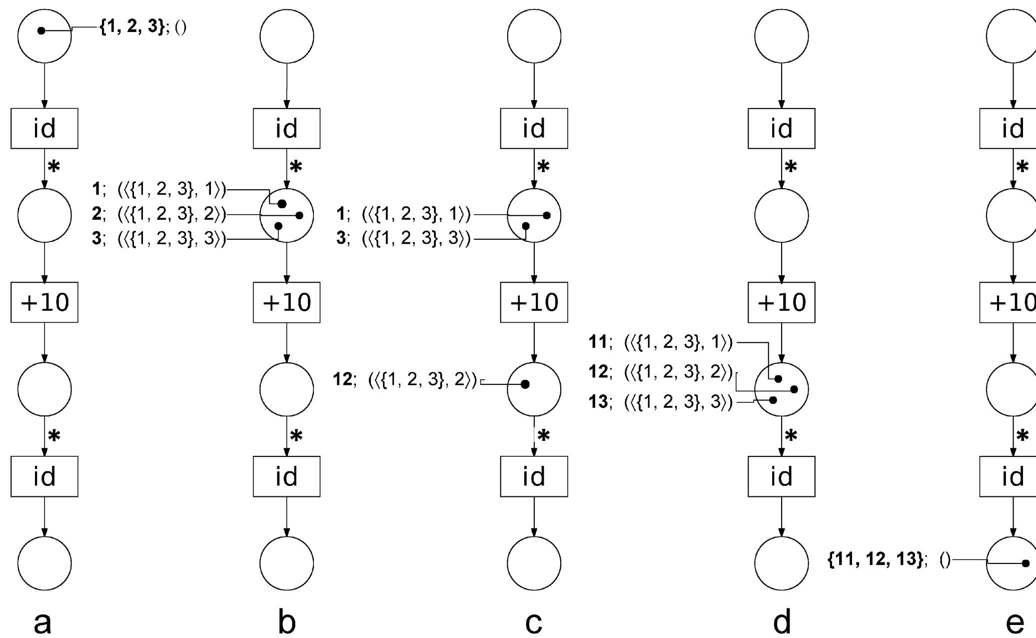


Fig. 3. An illustration of the unnest/nest edges and the unnesting history.

empty history. When the upper *id* transition fires, a token for each element of the output value $\{1, 2, 3\}$ is produced as shown in (b). The history is extended at the end with a pair that contains, first, the set that was unnested and, second, the element for which this particular token was produced. As is shown in (b)–(d) transitions without any unnest or nest edge 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 bottom *id* transition, as is shown in (d), it can fire and combine them into a single set-valued token as is shown in (e). A transition can verify if all the tokens that belong to the same unnesting group have arrived by looking at their histories. Note that where the firing of a transition with an unnest edge adds a pair to the history, firing a transition with a nest edge removes a pair from the history.

The second example (see Fig. 4) presents what happens when one transition has unnest/nest edges as well as normal edges. The initial state is presented in (a). As shown in (b), after firing transition *id*, the token representing an empty set has been consumed. Since *id* has an unnest edge, the result of its computation—an empty set—has been unnested and zero tokens have been inserted into the right output place. Yet the left output place is connected by a normal edge and a token has been produced there. Because unnesting has been performed on the "$*$" annotated edges, its history has been extended with a pair consisting of twice the unnested set. After some additional processing this token transports a set of three numbers $\{1, 2, 3\}$ as can be observed in (c). Now the set union transition can fire. Although one of its input places is empty, it is enabled because it is connected by a nest edge and the examination of the history of the token from the other input place that was connected by a normal edge shows that tokens representing elements of an empty set are to be expected there (so no tokens need to be consumed). When the set union transition fires, a set of $\{1, 2, 3\}$ will be produced as a result of the union of $\{1, 2, 3\}$ with an empty set. As is shown in (d) another unnest can be performed and this time tokens are inserted to both output places.

In the case of transitions with many input edges tokens consumed from distinct input places must either have the same history or must represent the same set. This way the history of the tokens produced by such a transition can be unambiguously determined, tokens representing elements of different sets do not interfere with each other in the body of the iteration and at the same time the order of execution is free of any unnecessary restrictions. This is illustrated on the third example (see Fig. 5). The transition in (a) can fire only if $h_1 = h_3$ or $h_2 = h_3$. Otherwise it is not enabled even though some tokens are in both of its input places. The transition in (b) can fire consuming tokens with values x_1 and x_2 from the left input place and x_3

Author's personal copy

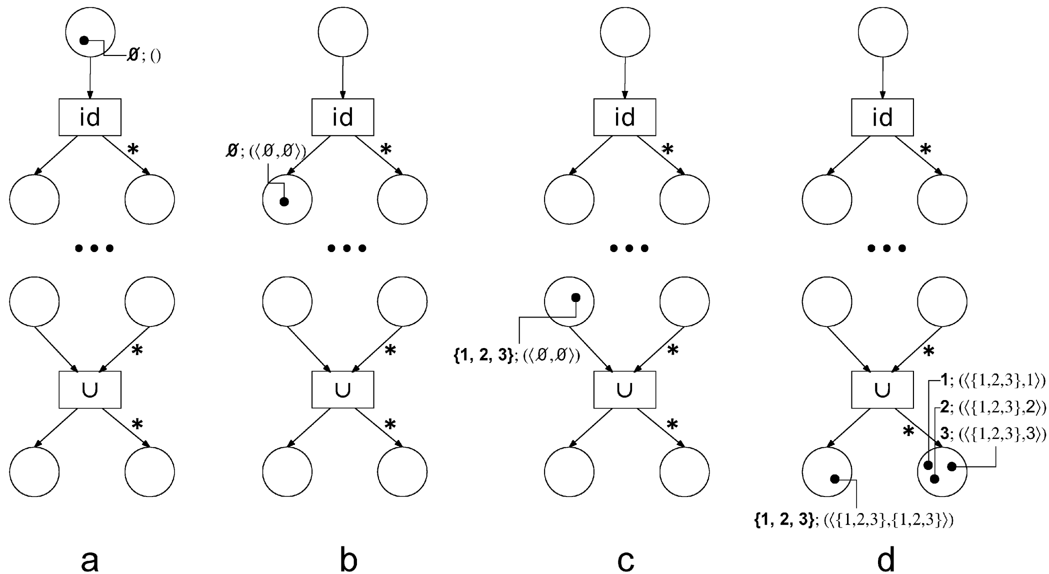


Fig. 4. An illustration of the unnesting history and iteration over empty sets.

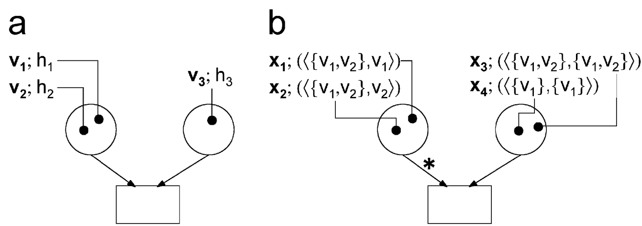


Fig. 5. An illustration of how history affects transitions with many input places.

from the right input place since they represent the same sets. A token with value x_4 cannot be consumed in this state, because there is no token representing the element of set $\{v_1\}$ in the left input place.

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 $x \in s$ or $x = s$. To manipulate histories we will use the following notation for extending a sequence with an element $(a_1, a_2, \ldots, a_n) \oplus a_{n+1} {:=} (a_1, a_2, \ldots, a_n, a_{n+1})$.

The fourth and final example presents why the whole history and not only its last element is taken into account while nesting. The dataflow in Fig. 6 unnest the input set of type $\{\langle v : \{integer\}, b : boolean \rangle\}$ and processes each of its pair values based on the boolean element. For pairs with a true value, every element of the associated set of integers is increased by one, while for pairs with a false value, the elements are decreased by one. In (a) the initial state with the input value is presented. In (b) the input value has been already unnested and, similarly as with the If–then–else example from Section 2.6, the paired elements have been separated to make the trueness based test. The $\langle \cdot, \cdot \rangle \circ \pi[v]$ transitions are used to dispose of the boolean value by creating a pair and projecting the boolean value out. Although in this example it is not important, since both pairs contained the same set $\{1, 2\}$, the transitions labeled $\langle \cdot, \cdot \rangle \circ \pi[v]$ would not consume values with different histories thus retaining the original pairing. In (c) the integer sets have been unnested and their values have been increased in the upper branch and decreased in the lower branch. The processed values are gathered in one place and are ready to be nested back. Observe that inspecting the last element of the history during nesting is not enough and the whole history has to be taken into account to prevent intermixing of the values processed by the lower and the upper branch.

It should be noted that our approach does not enforce iterating over elements of a set in any particular order and the transition semantics is local, yet it is always possible to determine if a given transition can fire and even in the case of a nested iteration over nested sets, tokens representing elements of different sets will not become intermixed.

4.2. Semantics of transitions

We define the semantics as a transition system, where the states are the distributions of tokens over

Author's personal copy

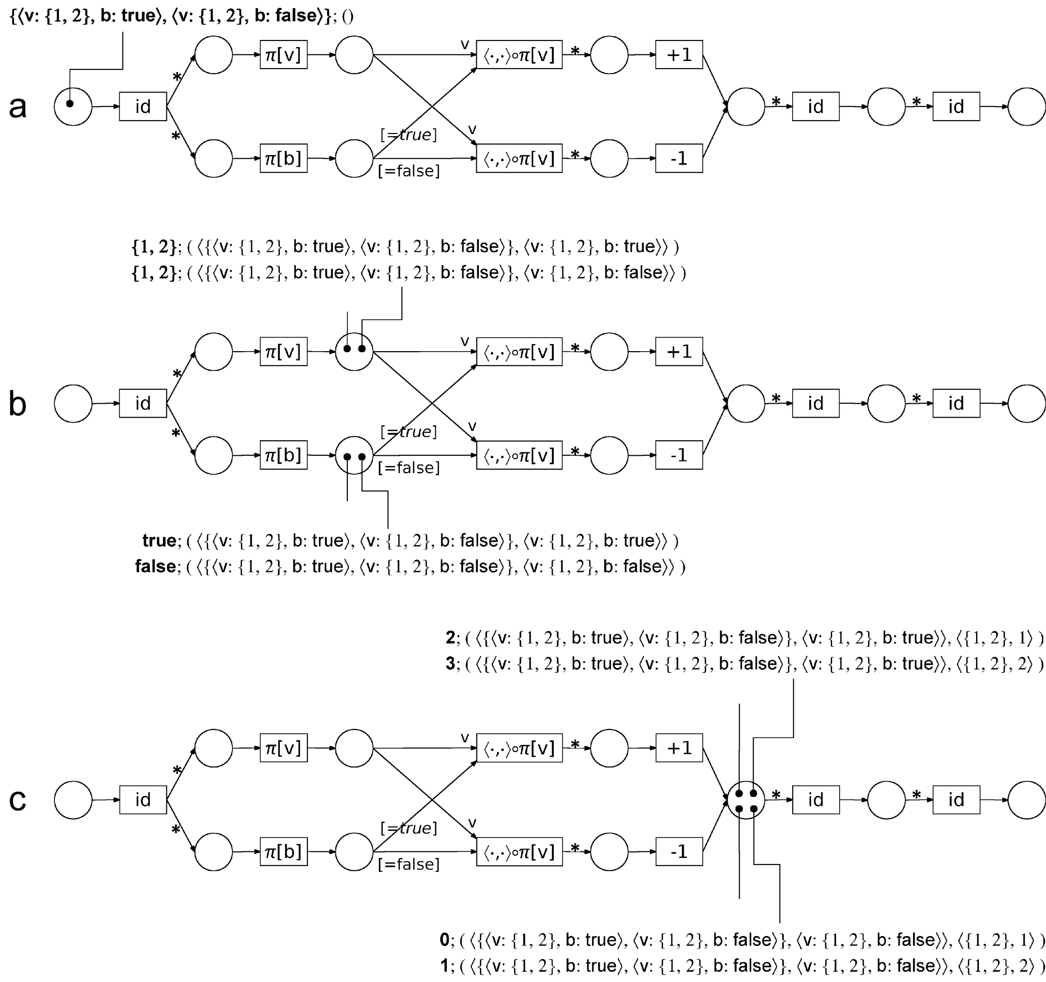


Fig. 6. An illustration of a nested iteration.

places and state changes are caused by firing enabled transitions. A transition is *enabled* in a given state, if from each of its input places it can consume tokens with matching histories—an arbitrary number from places connected by nest edges or one if it is not the case. Those tokens/sets of tokens represent values that will become arguments for the function represented by the enabled transition. The choice of such tokens and the function arguments determined by it are called an *enabling configuration*.

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:

$$\begin{aligned}\langle v,h\rangle \curvearrowright e = {} & (EA(e)=\varepsilon) \vee (EA(e) = \text{“}*\text{”}) \\ & \vee (EA(e) = \text{“} = \text{true”} \wedge v = \text{true}) \\ & \vee (EA(e) = \text{“} = \text{false”} \wedge v = \text{false}) \\ & \vee (EA(e) = \text{“} = \emptyset\text{”} \wedge v = \emptyset) \\ & \vee (EA(e) = \text{“}\neq\emptyset\text{”} \wedge v \neq \emptyset).\end{aligned}$$

Definition 7 (*Enabling configuration*). Given a transition t in marking M, an enabling configuration is a function $EC : \bullet t \rightarrow 2^K$ such that:

(i) for all places $p \in \bullet t$ and for all tokens $k \in EC(p)$ it holds that $M(p,k) \geqslant 1$ and $k \curvearrowright \langle p,t\rangle$,
(ii) at least one token is in the range of EC, i.e., $\bigcup_{p\in\bullet t} EC(p) \neq \emptyset$, and
(iii) there is a history h such that:
- if t has at least one nest edge, then there exists a set $S = \{x_1,\ldots,x_m\} \in CV$ such that for all places $p \in \bullet t$ it holds that

$$EC(p) = \begin{cases} \{\langle v_{p,1}, h \oplus \langle S, x_1\rangle\rangle,\ldots,\langle v_{p,m}, h\oplus\langle S,x_m\rangle\rangle\} \\ \qquad \text{if } EA(\langle p,t\rangle) = \text{“}*\text{”}, \\ \{\langle v_p, h \oplus \langle S,S\rangle\rangle\} \\ \qquad \text{if } EA(\langle p,t\rangle) \neq \text{“}*\text{”} \end{cases}$$

Author's personal copy

for some complex values $v_{p,1}, \ldots, v_{p,m}$ and v_p,

- if t has no nest edge, then for all places $p \in \bullet t$ it holds that $EC(p) = \{\langle v_p, h \rangle\}$ for some complex value v_p.

Note that since the range of the enabling configuration contains at least one token, it holds that if such an EC exists, then h is uniquely determined, so we denote it as h_{EC}.

Moreover, given such an EC we define the enabling configuration value function $ECV_{EC} : \bullet t \to CV$, which with a place p associates the value represented by the tokens pointed by $EC(p)$, i.e., for all places $p \in \bullet t$ it holds that

$$ECV_{EC}(p) = \begin{cases} \{v_{p,1}, \ldots, v_{p,m}\} & \text{if } EA(\langle p, t \rangle) = \text{“}*\text{”}, \\ v_p & \text{if } EA(\langle p, t \rangle) \neq \text{“}*\text{”}. \end{cases}$$

A transition for which an enabling configuration exists can fire and it is called *enabled*. In a given state many enabling configurations can exist for one transition. For example, if t has two input places connected by normal edges, one of its input place contains two tokens, the other contains three tokens and all the tokens have the same history, then there exist six enabling configurations for t in this state.

Definition 8 (*Enabled transition*). Transition t is enabled in a given marking M if and only if there exists an enabling configuration for t in M.

When a transition fires, it consumes tokens according to some enabling configuration EC and the transition's associated function is being computed with the arguments pointed by ECV_{EC}.

4.2.1. State transition (firing a transition)

For each $t \in T$ it holds that $M_1 \xrightarrow{t} M_2$ if and only if there exists an enabling configuration EC for t in marking M_1 such that:

(1) for all places $p \in \bullet t$ it holds that:
 (a) $M_2(p, k) = M_1(p, k) - 1$ if $k \in EC(p)$, and
 (b) $M_2(p, k) = M_1(p, k)$ if $k \notin EC(p)$;

(2) if t has no unnest edges, then for all places $p \in t\bullet$ it holds that, if v_{res} is the result of $\Phi_{TN(t)}(\langle l_1 : v_1, \ldots, l_n : v_n \rangle)$, in case when $\Phi_{TN(t)}$ is a deterministic function, or one of its possible results, when it is non-deterministic, where $\{\langle l_1, v_1 \rangle, \ldots, \langle l_n, v_n \rangle\} = \{\langle EN(\langle p', t \rangle), ECV_{EC}(p') \rangle \mid p' \in \bullet t\}$ then:
 (a) $M_2(p, \langle v_{res}, h_{EC} \rangle) = M_1(p, \langle v_{res}, h_{EC} \rangle) + 1$, and
 (b) $M_2(p, \langle v', h' \rangle) = M_1(p, \langle v', h' \rangle)$ if $\langle v', h' \rangle \neq \langle v_{res}, h_{EC} \rangle$;

(3) if t has at least one unnest edge, then for all places $p \in t\bullet$ it holds that, if v_{res} is the result of $\Phi_{TN(t)}(\langle l_1 : v_1, \ldots, l_n : v_n \rangle)$, in case when $\Phi_{TN(t)}$ is a deterministic function, or one of its possible results, when it is non-deterministic, where $\{\langle l_1, v_1 \rangle, \ldots, \langle l_n, v_n \rangle\} = \{\langle EN(\langle p', t \rangle), ECV_{EC}(p') \rangle \mid p' \in \bullet t\}$ then:
 (a) $M_2(p, \langle v_{res}, h_{EC} \oplus \langle v_{res}, v_{res} \rangle \rangle) = M_1(p, \langle v_{res}, h_{EC} \oplus \langle v_{res}, v_{res} \rangle \rangle) + 1$ if $EA(\langle t, p \rangle) \neq$ “$*$”, and
 (b) $M_2(p, \langle v', h' \rangle) = M_1(p, \langle v', h' \rangle)$ if $EA(\langle t, p \rangle) \neq$ “$*$” and $\langle v', h' \rangle \neq \langle v_{res}, h_{EC} \oplus \langle v_{res}, v_{res} \rangle \rangle$
 (c) $M_2(p, \langle v, h_{EC} \oplus \langle v_{res}, v \rangle \rangle) = M_1(p, \langle v, h_{EC} \oplus \langle v_{res}, v \rangle \rangle) + 1$ if $EA(\langle t, p \rangle) =$ “$*$” and $v \in v_{res}$, and
 (d) $M_2(p, \langle v', h' \rangle) = M_1(p, \langle v', h' \rangle)$ if $EA(\langle t, p \rangle) =$ “$*$” and $\langle v', h' \rangle \neq \langle v, h_{EC} \oplus \langle v_{res}, v \rangle \rangle$ for all $v \in v_{res}$;

(4) for all places $p \notin \bullet t \cup t\bullet$ it holds that $M_2(p, k) = M_1(p, k)$ for all tokens $k \in K$.

It should be noted that for a given state M_1, a transition t and two not equal states M_2 and M_3 it can hold that $M_1 \xrightarrow{t} M_2$ and $M_1 \xrightarrow{t} M_3$. This is because in M_1 there can be more than one enabling configuration for t. It can also be the case that the function represented by t is not a deterministic one and transitions to M_2 and M_3 are possible for the same enabling configuration, because two different output values can be produced.

We adopt the following Petri net notations:

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

Author's personal copy

or more enabled transitions may fire concurrently, if there are enough input tokens for both of them.

5. A bioinformatics dataflow example

In this section we present a dataflow based on a part of a real bioinformatics example [15]. The dataflow is shown in Fig. 7. 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

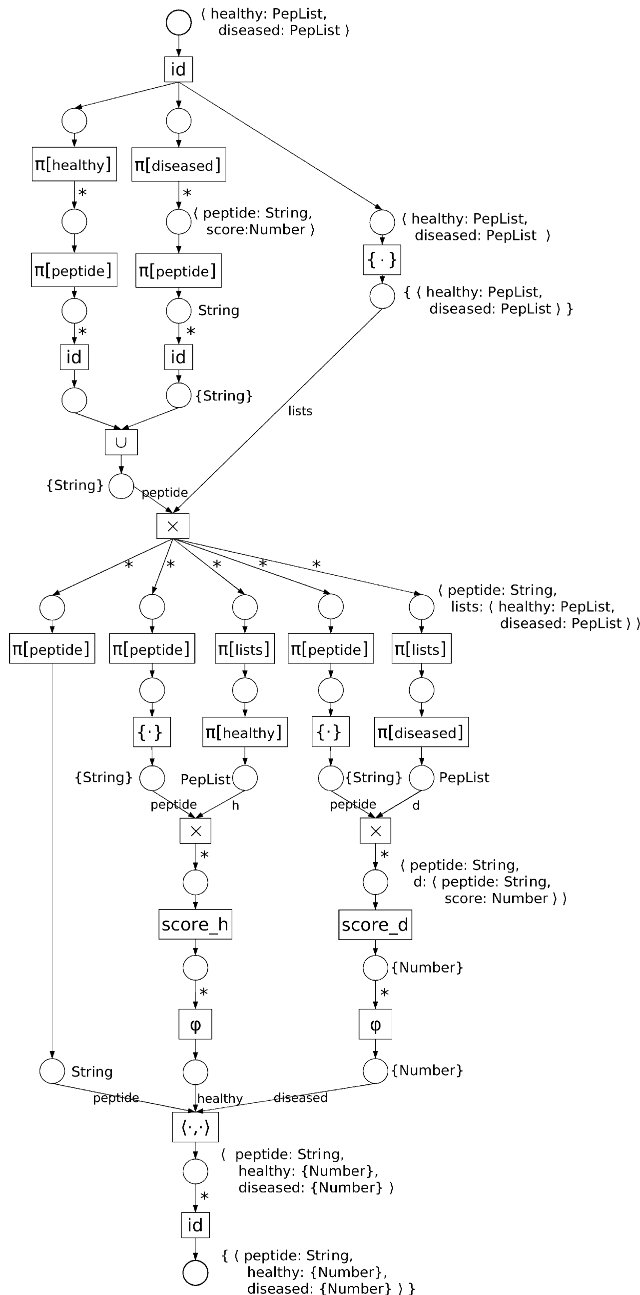


Fig. 7. Finding differences in peptide content of two samples.

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 $\langle$ `healthy`: *PepList*, `diseased` :*PepList* $\rangle$ with complex type *PepList* = $\{\langle$`peptide`: **String**, `score`: **Number**$\rangle\}$. Each data set contains tuples consisting of an identified peptide, represented by the basic type **String**, and the associated confidence score, represented by the basic 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 an 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: $\{\langle$`peptide`: **String**, `healthy`: $\{$**Number**$\}$, `diseased`: $\{$**Number**$\}\rangle\}$.

The global structure of the dataflow can be described as follows. In the first part up to and including the first transition labeled $\times$ it computes the Cartesian product of two sets. The first set is computed in the left branch, which consists again of two sub-branches, and is the union of all mentioned peptides in the initial tuple. The second set is computed in the right branch and is the singleton set containing the initial tuple. In the second part of the workflow, between the first Cartesian product and the final place, the workflow iterates over the result of the first part and processes the tuples in the Cartesian product in parallel in three branches, where the rightmost two branches themselves consist of two sub-branches, and combines their results into a single tuple with a tuple constructor. The first branch simply projects on the peptide in tuple. The second and third branch compute the scores of this peptide in the "healthy" peptide list and the "diseased" peptide list, respectively. They do so by computing the Cartesian product of the peptide and the relevant peptide list, iterating over the result and applying to each tuple the function score_h (or score_d) which compares the first peptide with the peptide in the nested tuple and if they are equal returns a singleton set with the score or an empty set otherwise. Note that the transitions labeled score_h and score_d could have been

Author's personal copy

decomposed further and replaced with dataflows, but are represented here by single transitions for brevity. Finally the dataflow collects all the tuples consisting of the peptide and its scores in the "healthy" and the "diseased" peptide list, into a single set.

6. Hierarchical dataflows

Our extension of workflow nets allows the reuse of various technical and theoretical results that are known about them. This is what we intend to demonstrate here by discussing a way of constructing workflows that guarantees that they always satisfy certain correctness criteria. A well-known technique for this is the use of refinement rules that allow the step-wise generation of Petri nets by replacing a transition or place with a slightly bigger net. Such refinement rules were studied by Berthelot in [16] and Murata in [8] as reduction rules that preserve liveness and boundedness properties of Petri nets. They are used by van der Aalst in [17], by Reijers in [18] and by Chrzastowski-Wachtel et al. in [19] to generate workflow nets. We show that the same principles can be applied to our extended notion of workflow net, and can be adapted to deal with the new problem of data-dependent control flow.

DFL 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, the result should appear in its sink. A special notation is introduced to distinguish between two state families.

Definition 9 (*Input state*). Given dataflow $D = \langle DFN, EN, TN, EA, PT \rangle$ with $DFN = \langle P, T, E, source, sink \rangle$ and value $v : PT(source)$ we define the *input state* $input_v^D$ as a marking such that:

- $input_v^D(source, \langle v, () \rangle) = 1$, and
- for all places $p \in P$ and tokens $k \in K$ such that $\langle p, k \rangle \neq \langle source, \langle v, () \rangle \rangle$ it holds that $input_v^D(p, k) = 0$.

Definition 10 (*Output state*). Given dataflow $D = \langle DFN, EN, TN, EA, PT \rangle$ with $DFN = \langle P, T, E, source, sink \rangle$ and value $v : PT(sink)$ we define the *output state* $output_v^D$ as a marking such that:

- $output_v^D(sink, \langle v, () \rangle) = 1$, and
- for all places $p \in P$ and tokens $k \in K$ such that $\langle p, k \rangle \neq \langle sink, \langle v, () \rangle \rangle$ it holds that $output_v^D(p, k) = 0$.

Starting with one token in the source and executing the dataflow need not always produce a result in the form of a single 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. For other dataflows the result token may be produced, but there still may be tokens left in other places. Furthermore, for some dataflows reaching a state in which there are no tokens at all is possible.

Examples of dataflows for which starting with one token does not always produce a result in the form of a single token in the sink place are shown in Fig. 8. For the dataflow (a) the token from the source can be consumed by a transition t_1 or t_2, but not by both of them at the same time. Transition t_3

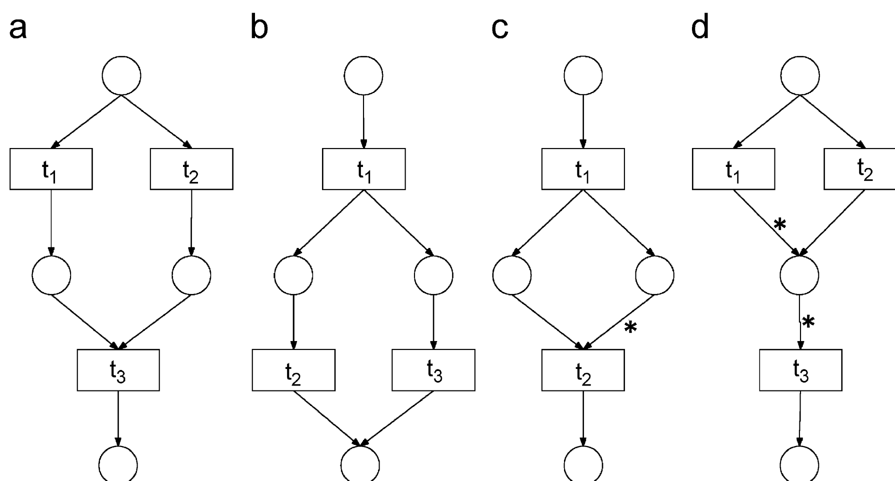


Fig. 8. Dataflows that may not finish properly.

Author's personal copy

will not become enabled then, because one of its input places will stay empty. The (b) case presents an opposite scenario. Transition t_1 produces two output tokens and after either t_2 or t_3 consumes one of them and produces a computation result, the second token is still there and another computation result can be produced. In the (c) case t_2 will never become enabled, since the tokens with history appropriate for nesting will never be produced by t_1. Similarly in case (d) if t_2 gets the source token, t_3 will not become enabled, because only t_1 can produce a token with the required history. But for (d) it may even be not enough, when the t_1 consumes the source token. If the source token carried an empty set, then in the resulting state all places would be empty.

Similar problems were also studied in the context of procedures modeled by classical workflow nets. The procedures without such problems are called *sound* [1]. A workflow net is considered to be sound if an only if:

(1) if a token is inserted into the sink, then there are no other tokens left,
(2) the computation can always be completed, that is, if one starts with a single token in the source and regardless of how the computation proceeds at start, it is always possible to reach a state with the only token in the sink place, and
(3) every transition can be fired, if one starts with a single token in the source.

This classical notion of soundness can be directly applied to dataflows such as (a) and (b) in Fig. 8 where the control flow does not depend upon the data, but in dataflows such as (c) and (d) where the control flow may depend upon the values and the unnesting histories associated with a token the notion needs to be adapted. Here tokens carry values, so there are many possible input states from which a computation can be started—one for each possible value for the first token. It is natural to require that each transition becomes enabled in some input state, but not in all.

Definition 11 (*Soundness*). A dataflow $D = \langle DFN, EN, TN, EA, PT \rangle$ with $DFN = \langle P, T, E, source, sink \rangle$ is sound if and only if:

(i) for each value $v' : PT(source)$ and every marking M such that $input^D_{v'} \overset{*}{\rightarrow} M$, if for some value $v'' : PT(sink)$ and history $h'' \in H$ it holds that $M(sink, \langle v'', h'' \rangle) > 0$, then $M = output^D_{v''}$,
(ii) for each value $v' : PT(source)$ and every marking M such that $input^D_{v'} \overset{*}{\rightarrow} M$ there exists a value $v'' : PT(sink)$ such that $M \overset{*}{\rightarrow} output^D_{v''}$, and
(iii) for each transition $t \in T$ there exists a value $v' : PT(source)$ and two markings M and M' such that $input^D_{v'} \overset{*}{\rightarrow} M \overset{t}{\rightarrow} M'$.

Although it seems desirable to require soundness of dataflows, many of the systems with conditional behavior will not satisfy (iii). The problem is often not caused by the structure of the net, but by operations associated with transition labels that are being used. An appearance of a value that activates some part of the net may be dependent on the value with which the dataflow is initiated. Checking if the right value can appear would be undecidable as is determining if an NRC expression returns an empty set. Indeed, it is well known that NRC can simulate the relational algebra [7]. That is why we introduce a weaker *semi-soundness* notion:

Definition 12 (*Semi-soundness*). A dataflow $D = \langle DFN, EN, TN, EA, PT \rangle$ with $DFN = \langle P, T, E, source, sink \rangle$ is semi-sound if and only if:

(i) for each value $v' : PT(source)$ and every marking M such that $input^D_{v'} \overset{*}{\rightarrow} M$, if for some value $v'' : PT(sink)$ and history $h'' \in H$ it holds that $M(sink, \langle v'', h'' \rangle) > 0$, then $M = output^D_{v''}$, and
(ii) for each value $v' : PT(source)$ and every marking M such that $input^D_{v'} \overset{*}{\rightarrow} M$ there exists a value $v'' : PT(sink)$ such that $M \overset{*}{\rightarrow} output^D_{v''}$.

6.1. Refinement rules

In this section we introduce refinement rules for generating what may be considered a well-structured dataflow. As we will show later, all dataflows generated in this way are semi-sound. By starting from a single place and applying the rules in a top-down manner we generate *blank dataflows*—dataflows without edge and transition naming. We call such generated blank dataflows *hierarchical blank dataflows*. From these we then obtain dataflows by adding edge and transition naming functions. These will be called *hierarchical dataflows*.

Definition 13 (*Blank dataflow*). A blank dataflow is a tuple $\langle DFN, EA \rangle$ where:

- $DFN = \langle P, T, E, source, sink \rangle$ is a dataflow net,

Author's personal copy

- $EA : (\circ T \rightarrow \{\text{``}=\text{true''}, \text{``}=\text{false''}, \text{``}=\emptyset\text{''}, \text{``}\neq\emptyset\text{''}, \text{``}*\text{''}, \varepsilon\}) \cup (\circ P \rightarrow \{\text{``}*\text{''}, \varepsilon\})$ is an edge annotation function.

The refinement rules are presented in Fig. 9. Each refinement replaces a subgraph presented on the left-hand side of the rule by the right-hand side one. The edge annotation for the replaced subgraph and the subgraph that it is replaced with is exactly as indicated. For each rule we define the concepts of *input nodes*, *output nodes* and *body nodes* as indicated in Table 2.

The right-hand side subgraph is connected to the rest of the blank dataflow as follows:

- All the incoming edges of the left-hand side input node are reconnected to all the input nodes of the right-hand side. The annotations are preserved. A visualization is presented in Fig. 10.
- For rules d and e all the remaining, i.e., not shown in the rule, outgoing edges of the input nodes on the left-hand side are reconnected to the input nodes on the right-hand side. The annotations are preserved. A visualization is presented in Fig. 11.
- All the outgoing edges of the left-hand side output node are reconnected to all the output nodes of the right-hand side. The annotations are preserved, with the exception that for rule f condition annotations are preserved only for outgoing edges of the node b_1 and outgoing edges of node b_2 are not annotated with conditions. A visualization is presented in Fig. 10.
- For rules d and e all the remaining, i.e., not shown in the rule, incoming edges of the output nodes on the left-hand side are reconnected to the output nodes on the right-hand side. The annotations are preserved. A visualization is presented in Fig. 11.
- All the incoming and outgoing edges of the left-hand side body nodes are reconnected to all the right-hand side body nodes. The annotations are preserved. A visualization is presented in Fig. 11.

There are certain preconditions that must hold when the rules are applied:

(i) For rule d to be applied, all the transitions in $\bullet a_1$ that are connected with a_1 by a non-annotated edge cannot have any unnest edges, i.e., for all $t \in \bullet a_1$ it holds that: if $EA(\langle t, a_1 \rangle) = \varepsilon$, then for all $p \in t\bullet$ it holds that $EA(\langle t, p \rangle) \neq \text{``}*\text{''}$.

(ii) For rule d to be applied, all the transitions in $\bullet a_1$ cannot have any other output places that are connected by an edge annotated in the same way and on which an *emptiness based decision* is

Table 2
Input, output and body nodes

	Rule a	Rule b	Rule c	Rule d	Rule e	Rule f
Input nodes	a, b_1	a, b_1	a, b_1	a_1, b_1	a_1, b_1	a, b_1, b_2
Output nodes	a, b_3	a, b_3	a, b_4	a_3, b_4	a_3, b_4	a, b_1, b_2
Body nodes	b_2	b_2	b_2, b_3	a_2, b_2, b_3	a_2, b_2, b_3	

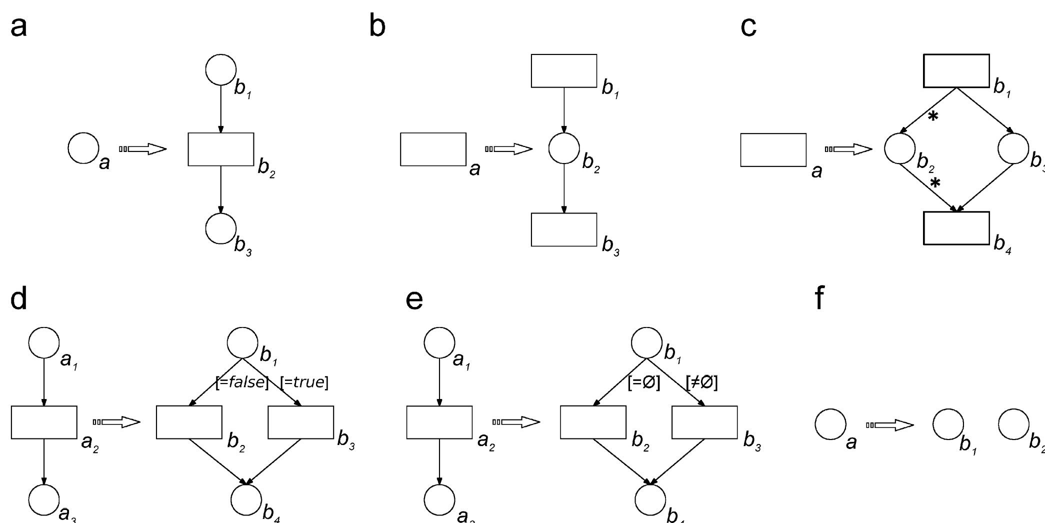


Fig. 9. Refinement rules. (a) sequential place split, (b) sequential transition split, (c) iteration split, (d) trueness based decision, (e) emptiness based decision, (f) AND-split.

Author's personal copy

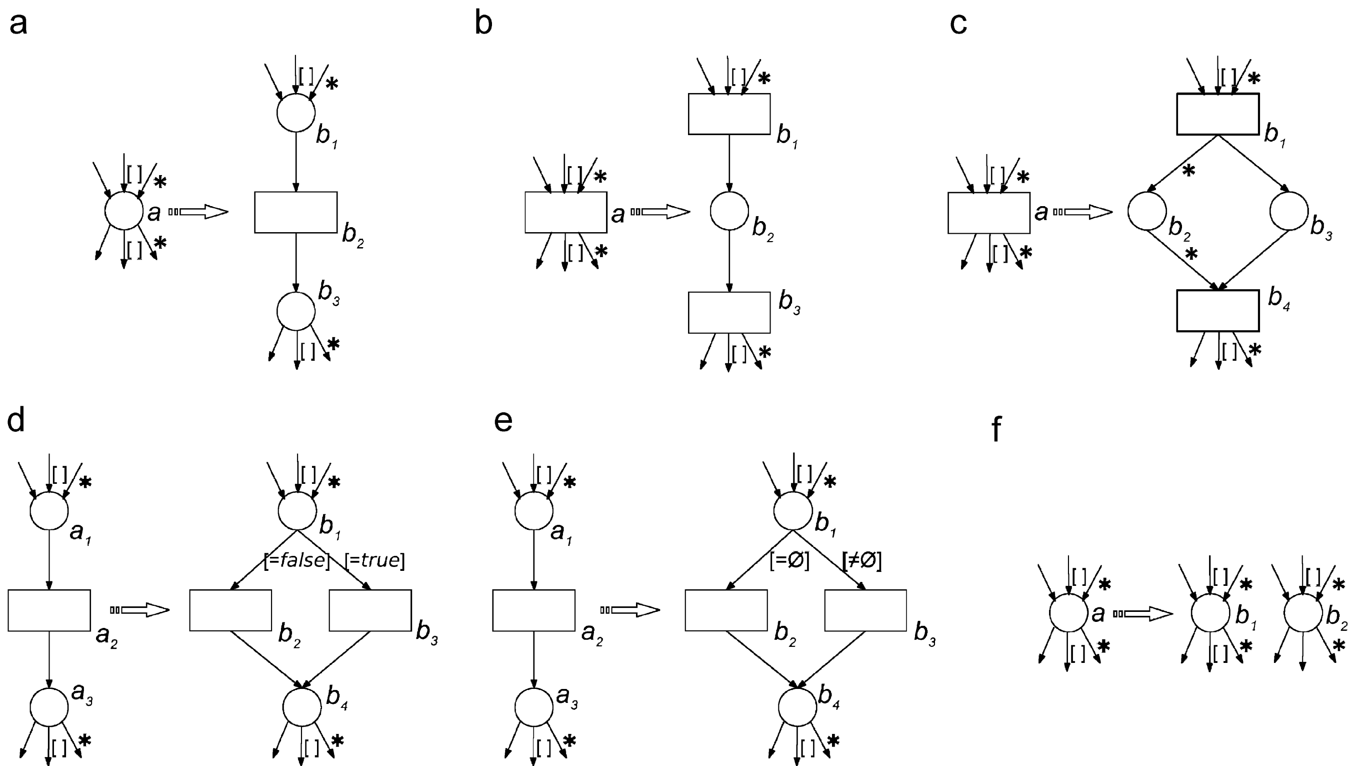


Fig. 10. Reconnecting of subgraphs. (a) sequential place split, (b) sequential transition split, (c) iteration split, (d) trueness based decision, (e) emptiness based decision, (f) AND-split.

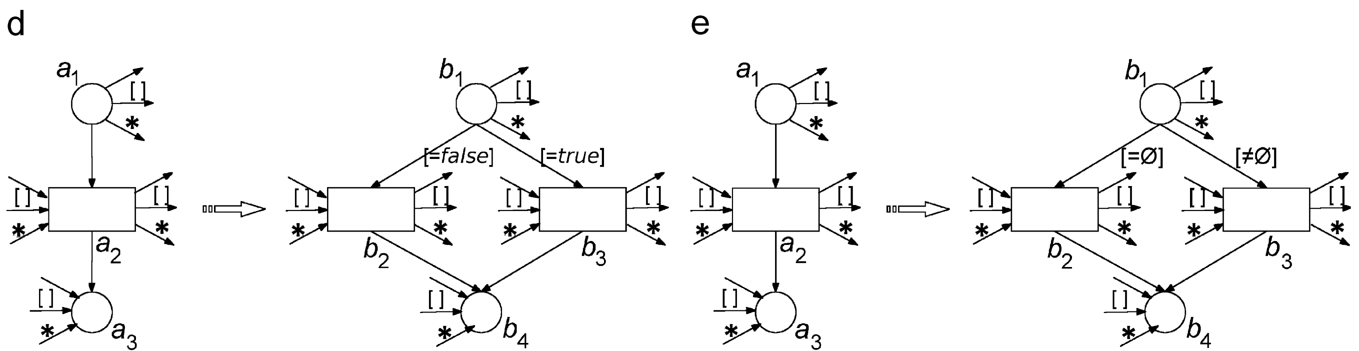


Fig. 11. Reconnecting of subgraphs—additional edges for rules d and e. (d) trueness based decision, (e) emptiness based decision.

performed, i.e., for all $t \in \bullet a_1$ and for all $p \in t\bullet$ it holds that: if $p \neq a_1$ and $EA(\langle t, a_1 \rangle) = EA(\langle t, p \rangle)$, then for all $t' \in p\bullet$ it holds that $EA(\langle p, t' \rangle) \notin \{\text{“} = \emptyset\text{”}, \text{“} \neq \emptyset\text{”}\}$.

(iii) For rule e to be applied, all the transitions in $\bullet a_1$ cannot have any other output places that are connected by an edge annotated in the same way and on which a *trueness based decision* is performed, i.e., for all $t \in \bullet a_1$ and for all $p \in t\bullet$ it holds that: if $p \neq a_1$ and $EA(\langle t, a_1 \rangle) = EA(\langle t, p \rangle)$, then it holds that for all $t' \in p\bullet$ it holds that $EA(\langle p, t' \rangle) \notin \{\text{“} = \text{true”}, \text{“} = \text{false”}\}$.

(iv) For rule f to be applied, a hasto have at least one incoming and one outgoing edge.

The first three preconditions are necessary so that it is always possible to label the generated blank dataflow such that it becomes a legal dataflow. (i) deals with a requirement that a token representing set values cannot be used to make a *trueness based decision* (see Fig. 12(i)), while (ii) and (iii) prevent using tokens with the same values in different kinds of tests (see Fig. 12(ii) and (iii)). Precondition (iv) guarantees that there is exactly one input and output place.

Author's personal copy

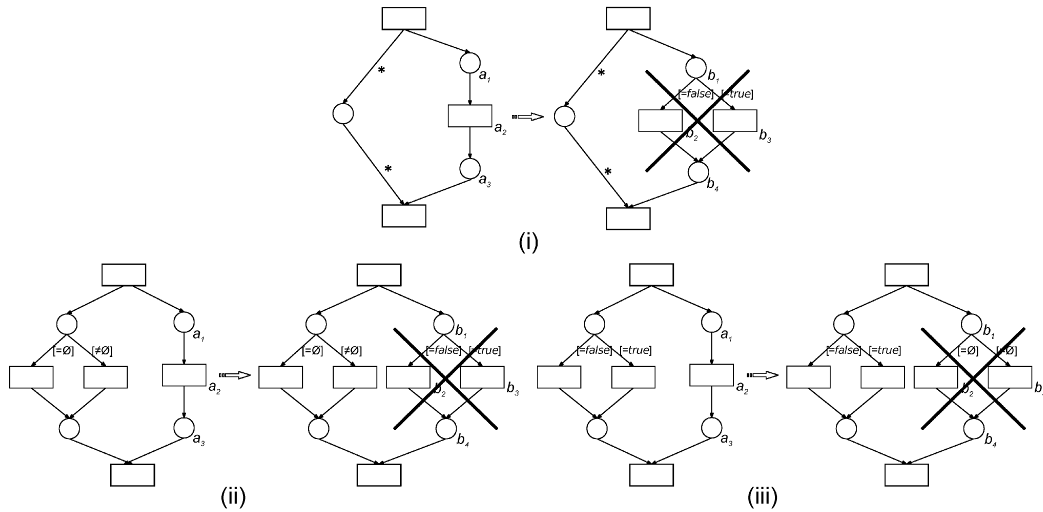


Fig. 12. Preconditions.

Definition 14 (*Hierarchical blank dataflow*). A blank dataflow which is obtained by starting with a blank dataflow that consists of a single place with no transitions and performing the transformations presented in Fig. 9 is called a hierarchical blank dataflow.

Definition 15 (*Hierarchical dataflow*). A hierarchical dataflow is a legal dataflow $D = \langle DFN, EN, TN, EA, PT\rangle$ obtained by labeling transitions and edges in a hierarchical blank dataflow $BDF = \langle DFN, EA\rangle$.

The rules and the aim to make dataflows structured as in structured programming languages were inspired by the work done on workflow nets by Chrzastowski-Wachtel et al. [19].

An instance of a computation of a particular dataflow, which starts in some input state, will be called a *run*. We will represent it as a pair of two sequences. The first one will contain successive transitions that were fired and the second one subsequent states including the input state.

Definition 16 (*Run*). Let $D = \langle DFN, EN, TN, EA, PT\rangle$ be a dataflow with a dataflow net $DFN = \langle P, T, E, source, sink\rangle$. A sequence of transitions $t_1, \ldots, t_n \in T$ with a sequence of markings $M_0, \ldots, M_n$ of D, where M_0 is an input state, forms a run if and only if it holds that $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} M_n$.

The run will be denoted as $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} M_n$. If M_n is an output state of D, then we will call such a run *complete*.

For a run $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} M_n$, a place p and history h we define a delta of tokens in p after firing a given transition t_{i+1} in a state M_i:

$$\Delta_i(p,h) = \sum_{v\in CV} M_{i+1}(p, \langle v,h\rangle) - \sum_{v\in CV} M_i(p,\langle v,h\rangle).$$

We will also want to count tokens inserted to a place (since there are no cycles, during one transition tokens are never inserted to and consumed from a place at the same time):

$$\Delta_i^+(p,h) = \begin{cases} \Delta_i(p,h) & \text{if } \Delta_i(p,h) > 0, \\ 0 & \text{otherwise.} \end{cases}$$

The number of tokens with a given history h inserted into a place p during a run $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} M_n$ will be called a *trace* of p and defined as $Tr(p,h) = \sum_{i=0}^{n-1} \Delta_i^+(p,h)$.

Lemma 17. *For each hierarchical dataflow* $D = \langle DFN, EN, TN, EA, PT\rangle$ *with a dataflow net* $DFN = \langle P, T, E, source, sink\rangle$ *and for each run* $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} M_n$ *of dataflow* D*, the trace of each place is bounded by* 1*, i.e., it holds that*

$\forall_{h\in H}\forall_{p\in P} Tr(p,h) \leqslant 1.$

Theorem 18. *Every hierarchical dataflow is semi-sound.*

Proof of Lemma 17 and Theorem 18. We will prove Lemma 17 and Theorem 18 together, by induction on the number of refinements applied in the generation of the blank dataflow. During this proof we will assume that in TL there are labels representing all the NRC expressions on the available external functions.

Author's personal copy

For a hierarchical dataflow consisting of only one place, all runs have empty transition sequence and the state sequence consists of only one state, which is an input and an output state at the same time. Therefore such dataflow is semi-sound and the sum in Lemma 17 contains no elements, thus is equal 0.

Let us assume by mathematical induction that for each hierarchical dataflow $D_n = \langle DFN_n, EN_n, TN_n, EA_n, PT_n\rangle$ with a dataflow net $DFN_n = \langle P_n, T_n, E_n, source_n, sink_n\rangle$ whose hierarchical blank dataflow was generated in $n \geqslant 0$ refinements it holds that:

(1) for each run $M'_0 \xrightarrow{t_1} M'_1 \xrightarrow{t_2} \cdots \xrightarrow{t_d} M'_d$ of D_n every trace of every place is bounded by 1,
(2) for each value $v' : PT_n(source_n)$ and marking M' such that $input^{D_n}_{v'} \xrightarrow{*} M'$, if for some value $v'' : PT_n(sink_n)$ and history $h'' \in H$ it holds that $M'(sink_n, \langle v'', h''\rangle) > 0$, then $M' = output^{D_n}_{v''}$, and
(3) for each value $v' : PT_n(source_n)$ and marking M' such that $input^{D_n}_{v'} \xrightarrow{*} M'$ there exists a value $v'' : PT_n(sink_n)$ such that $M' \xrightarrow{*} output^{D_n}_{v''}$.

We will show that if $D_{n+1} = \langle DFN_{n+1}, EN_{n+1}, TN_{n+1}, EA_{n+1}, PT_{n+1}\rangle$ with a dataflow net $DFN_{n+1} = \langle P_{n+1}, T_{n+1}, E_{n+1}, source_{n+1}, sink_{n+1}\rangle$ is an arbitrary hierarchical dataflow whose hierarchical blank dataflow was generated in $n+1$ refinements, then:

(i) for each run $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_m} M_m$ of D_{n+1} every trace of every place is bounded by 1,
(ii) for each value $v' : PT_{n+1}(source_{n+1})$ and each marking M such that $input^{D_{n+1}}_{v'} \xrightarrow{*} M$, if for some value $v'' : PT_{n+1}(sink_{n+1})$ and history $h'' \in H$ it holds that $M(sink_{n+1}, \langle v'', h''\rangle) > 0$, then $M = output^{D_{n+1}}_{v''}$, and
(iii) for each value $v' : PT_{n+1}(source_{n+1})$ and each marking M such that $input^{D_{n+1}}_{v'} \xrightarrow{*} M$ there exists a value $v'' : PT_{n+1}(sink_{n+1})$ such that $M \xrightarrow{*} output^{D_{n+1}}_{v''}$.

Let us consider each possible case for the last, $(n+1)$th, refinement applied.

(a) The last applied refinement was a *sequential place split* (see Fig. 9a). Let $BDF_n = \langle DFN_n, EA_n\rangle$ with a dataflow net $DFN_n = \langle P_n, T_n, E_n, source_n, sink_n\rangle$ be a blank hierarchical dataflow generated by the first n refinements that generated the blank dataflow of D_{n+1}. Let $D_n = \langle DFN_n, EN_n, TN_n, EA_n, PT_n\rangle$ be a hierarchical dataflow labeled accordingly to the labeling of D_{n+1}. Since there is no b_2 transition in D_n, to keep D_n legal, the function that it computes is incorporated into the transitions that follow it directly, if there are any, or is omitted otherwise. That is $PT_n(a) = PT_{n+1}(b_1)$ and for each $t \in a\bullet$ it holds that

$$TN_n(t) = \begin{cases} TN_{n+1}(t)\big|^{\Phi_{TN_{n+1}(b_2)}}_{EN_{n+1}(\langle b_3, t\rangle)} & \text{if } EA(\langle a,t\rangle) \neq \text{“}*\text{”}, \\ TN_{n+1}(t)\big|^{map(\Phi_{TN_{n+1}(b_2)})}_{EN_{n+1}(\langle b_3, t\rangle)} & \text{if } EA(\langle a,t\rangle) = \text{“}*\text{”}. \end{cases}$$

Here $tl|^f_{l_i}$ means the transition label obtained from the transition label tl, by letting the input from edge l_i through f first. Namely, if $IT(tl) = \langle l_1 : \tau_1, \ldots, l_k : \tau_k\rangle$ and $f : \tau'_i \to \tau_i$, then $IT(tl|^f_{l_i}) = \langle l_1 : \tau_1, \ldots, l_i : \tau'_i, \ldots, l_k : \tau_k\rangle$, $OT(tl|^f_{l_i}) = OT(tl)$ and for all values $v_1, \ldots, v_k$ of the appropriate types $\Phi_{tl|^f_{l_i}}(\langle l_1 : v_1, \ldots, l_k : v_k\rangle) = \Phi_{tl}(\langle l_1 : v_1, \ldots, l_i : f(v_i), \ldots, l_k : v_k\rangle)$.

For each run $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_m} M_m$ of D_{n+1} we define a corresponding run $M'_0 \xrightarrow{t'_1} M'_1 \xrightarrow{t'_2} \cdots \xrightarrow{t'_d} M'_d$ of D_n. The transitions are fired in the same order, they consume the same tokens and functions produce the same results, but all occurrences of b_2 are omitted. It is easy to see that $M'_0 \xrightarrow{t'_1} M'_1 \xrightarrow{t'_2} \cdots \xrightarrow{t'_d} M'_d$ is indeed a run of D_n and that it is unambiguously defined. Let us assume that the subsequence of not omitted transitions have indices $i_1, \ldots, i_d$. The markings of D_n are equal to their counterparts in D_{n+1} on all the places that appear in both of the dataflows (i.e., for every $p \in P_n \cap P_{n+1}$ and $k \in K$ it holds that $M_0(p,k) = M'_0(p,k)$ and $M_{i_1}(p,k) = M'_1(p,k), \ldots,$ $M_{i_d}(p,k) = M'_d(p,k)$). Whereas place a contains all the tokens that in the counterpart marking are stored in b_1 as well as all the tokens that were consumed from b_1 in order to produce the tokens that in the counterpart are stored in b_3. This correspondence in not an injection, though. For each run of D_n there can be many corresponding runs of D_{n+1}. This is because there is a choice when to fire b_2, if tokens inserted into a are not immediately consumed.

As for (i), the content of places in $M_0, M_{i_1}, \ldots, M_{i_d}$ is bounded by the content of places in $M'_0, M'_1, \ldots, M'_d$, respectively. In the remaining markings the only difference is that some tokens are consumed from b_1, processed by b_2 and the result is placed in b_3. Thus the traces of places in markings of D_{n+1} are limited by the traces of places in markings of D_n, for which the induction assumption holds.

As for (ii), we can assume without loss of generality that M_m is the first marking in $M_0, \ldots, M_m$ in which $sink_{n+1}$ is not empty. We will

Author's personal copy

first consider the case when $sink_{n+1} \neq b_3$ and $t_m \neq b_2$. In the M'_d of the corresponding run $sink_n$ is also not empty ($sink_n = sink_{n+1}$). From the induction assumption in M'_d there is only one token—the one in $sink_n$. Since in M_m there is the same number of tokens, then also in M_d there is only one token—the one in the $sink_{n+1}$. In the case where $sink_{n+1} = b_3$, it is only possible for a token to be inserted into $sink_{n+1} = b_3$, when there was a token to be consumed from b_1. Yet, when the first token is inserted into b_1, there are no other tokens since in the corresponding run a token is inserted into a, which is a sink there. Since $M_0, \ldots, M_m$ was arbitrarily chosen, (ii) holds.

As for (iii), let $v' : PT_{n+1}(source_{n+1})$ and let M be a marking of D_{n+1} such that $input_{v'}^{D_{n+1}} \xrightarrow{*} M$. By the definition of marking reachability there exists a run $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_m} M_m$, where $M_0 = input_{v'}^{D_{n+1}}$ and $M_m = M$. We know that for this run in D_n there exists a corresponding run $M'_0 \xrightarrow{t'_1} M'_1 \xrightarrow{t'_2} \cdots \xrightarrow{t'_d} M'_d$, where $M'_0 = input_{v'}^{D_n}$. From the semi-soundness of D_n it follows that for some $v'' : PT_n(sink_n)$ this corresponding run can be extended into a complete run $M'_0 \xrightarrow{t'_1} M'_1 \xrightarrow{t'_2} \cdots \xrightarrow{t'_d} M'_d \xrightarrow{t'_{d+1}} M'_{d+1} \xrightarrow{t'_{d+2}} \cdots \xrightarrow{t'_{d+q}} M'_{d+q}$, where $M'_{d+q} = output_{v''}^{D_n}$. For it in turn there exists a corresponding complete run $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_m} M_m \xrightarrow{t_{m+1}} M_{m+1} \xrightarrow{t_{m+2}} \cdots \xrightarrow{t_{m+r}} M_{m+r}$ in D_{n+1}, which at the beginning is identical to the run of D_{n+1} we started from and in M_{m+r} place b_1 is empty (if $b_3 = sink_{n+1}$, b_1 can be emptied by firing b_2). This completes the proof, since we have shown that $M_m \xrightarrow{*} output_{v'''}^{D_{n+1}}$, for

$$v''' = \begin{cases} \Phi_{TN_{n+1}(b_2)}(v'') & \text{if } a = sink_n, \\ v'' & \text{otherwise.} \end{cases}$$

(b) The last refinement was a *sequential transition split* (see Fig. 9b). As previously, with the first n refinements, we can construct a blank hierarchical dataflow and label it accordingly to the labeling of D_{n+1}. In the resulting dataflow D_n, the label of a represents the composition of functions $\Phi(TN_{n+1}(b_3))$ and $\Phi(TN_{n+1}(b_1))$. That is $IT_n(TN_n(a)) = IT_{n+1}(TN_{n+1}(b_1))$, $OT_n(TN_n(a)) = OT_{n+1}(TN_{n+1}(b_3))$ and $\Phi_{TN_n(a)} = \Phi_{TN_{n+1}(b_3)} \circ \Phi_{TN_{n+1}(b_1)}$.

For each run $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_m} M_m$ of D_{n+1} we define a corresponding run $M'_0 \xrightarrow{t'_1} M'_1 \xrightarrow{t'_2} \cdots \xrightarrow{t'_d} M'_d$ of D_n. The transitions are fired in the same order, they consume the same tokens and functions produce the same results, but all occurrences of b_1 are omitted and all occurrences of b_3 are replaced with a. It is easy to see that $M'_0 \xrightarrow{t'_1} M'_1 \xrightarrow{t'_2} \cdots \xrightarrow{t'_d} M'_d$ is indeed a run of D_n and that it is unambiguously defined. Let us assume that the subsequence of not omitted (other that b_1) transitions have indices $i_1, \ldots, i_d$. The markings of D_n are equal to their counterparts in D_{n+1} on all the places that appear in both of the dataflows except the ones in $\bullet a = \bullet b_1$ (i.e., for every $p \in ((P_n \cap P_{n+1}) \setminus \bullet a)$ and $k \in K$ it holds that $M_0(p,k) = M'_0(p,k)$ and $M_{i_1}(p,k) = M'_1(p,k), \ldots, M_{i_d}(p,k) = M'_d(p,k)$)). Whereas each place in $\bullet a$ contains all the tokens that in the counterpart marking are stored in the corresponding place in $\bullet b_1$ as well all the tokens that were consumed from that place in order to produce the tokens that are in the counterpart stored in b_2. This correspondence in not an injection, though. For each run of D_n there can be many corresponding runs of D_{n+1}. This is because there is a choice when to fire b_3, if tokens produced by a into $a\bullet$ are not immediately consumed.

The rest of the proof follows the one given for (a).

(c) The last refinement was an *iteration split* (see Fig. 9c). As previously, with the first n refinements, we can construct a blank hierarchical dataflow and label it accordingly to the labeling of D_{n+1}. In the resulting dataflow D_n, the label of transition a represents a composition of three functions: $\Phi_{TN_{n+1}(b_4)}$, a *pair* function of appropriate type that constructs a pair of twice its argument, and a function $\Phi_{TN_{n+1}(b_1)}$. That is $IT_n(TN_n(a)) = IT_{n+1}(TN_{n+1}(b_1))$, $OT_n(TN_n(a)) = OT_{n+1}(TN_{n+1}(b_4))$ and $\Phi_{TN_n(a)} = \Phi_{TN_{n+1}(b_4)} \circ \, pair \circ \Phi_{TN_{n+1}(b_1)}$. The correspondence of runs is analogous as in (b). The rest of the proof follows.

(d) The last refinement was a *trueness based decision* (see Fig. 9d). As previously, with the first n refinements, we can construct a blank hierarchical dataflow and label it accordingly to the labeling of D_{n+1}. That is $IT_n(TN_n(a_2)) = IT_{n+1}(TN_{n+1}(b_2)) = IT_{n+1}(TN_{n+1}(b_3))$ and $OT_n(TN_n(a_2)) = OT_{n+1}(TN_{n+1}(b_2)) = OT_{n+1}(TN_{n+1}(b_3))$, and for the edge names $EN_n(\langle a_1, a_2 \rangle) = EN_{n+1}(\langle b_1, b_2 \rangle) = EN_{n+1}(\langle b_1, b_3 \rangle)$. Assume $IT_{n+1}(TN_{n+1}(b_2)) = \langle l_1 : \tau_1, \ldots, l_k : \tau_k, EN_n(\langle a_1, a_2 \rangle) : PT(a_1) \rangle$. $TN_n(a_2)$ represents a function computing *if–then–else* expression that results in evaluating of either of $\Phi_{TN_{n+1}(b_2)}$ or $\Phi_{TN_{n+1}(b_3)}$. Which means that for every values $v_1, \ldots, v_k$ of appropriate types and every $v : PT(a_1)$

Author's personal copy

the result of function $\Phi_{TN_n(a_2)}(\langle l_1 : v_1, \ldots, l_k : v_k, EN_n(\langle a_1, a_2 \rangle) : v \rangle)$ equals $\Phi_{TN(b_2)}(\langle l_1 : v_1, \ldots, l_k : v_k, EN_n(\langle a_1, a_2 \rangle) : v \rangle)$, if $v = \text{false}$, or $\Phi_{TN(b_3)}(\langle l_1 : v_1, \ldots, l_k : v_k, EN_n(\langle a_1, a_2 \rangle) : v \rangle)$, otherwise.

The correspondence of runs in this case is a bijection. Transitions are fired in the same order, but all occurrences of b_2 and b_3 are replaced with a_2 or depending on the consumed token value a_2 is replaced by b_2 or b_3. The markings are equal to their counterparts in all the place that appear in both of the dataflows. Whereas a_1 contains the same tokens as b_1 and a_3 the same tokens as b_4.

The rest of the proof follows.

(e) The last refinement was an *emptiness based decision* (see Fig. 9e). The proof follows the one given for (d).

(f) The last refinement was an *AND-split* (see Fig. 9f). As previously, with the first n refinements, we can construct a blank hierarchical dataflow and label it accordingly to the labeling of D_{n+1}. Since *AND-split* was the last refinement applied in generation of blank dataflow of D_{n+1}, we know that $b_1\bullet = b_2\bullet$, $\bullet b_1 = \bullet b_2$ and thus $PT_{n+1}(b_1) = PT_{n+1}(b_2)$. For every transition $t_{n+1} \in b_1\bullet$, where $IT(TN_{n+1}(t_{n+1})) = \langle l_1 : \tau_1, \ldots, l_k : \tau_k, EN_{n+1}(\langle b_1, t_{n+1} \rangle) : PT_{n+1}(b_1), EN_{n+1}(\langle b_2, t_{n+1} \rangle) : PT_{n+1}(b_2) \rangle$, its corresponding transition $t_n \in a\bullet$ is defined as follows:

- $IT(TN_n(t_n)) = \langle l_1 : \tau_1, \ldots, l_k : \tau_k, EN_{n+1}(\langle b_1, t_{n+1} \rangle) : PT_{n+1}(b_1) \rangle$, that is $EN_n(\langle a, t_n \rangle) = EN_{n+1}\langle b_1, t_{n+1} \rangle$,
- $OT(TN_n(t_n)) = OT(TN_{n+1}(t_{n+1}))$,
- for all values $v_1, \ldots, v_k$ of appropriate types and all $v : PT_n$ the function computed by this transition is defined as follows $\Phi_{TN_n(t_n)}(\langle l_1 : v_1, \ldots, l_k : v_k, EN_{n+1}(\langle b_1, t_{n+1} \rangle) : v \rangle) = \Phi_{TN_{n+1}(t_{n+1})}(\langle l_1 : v_1, \ldots, l_k : v_k, EN_{n+1}(\langle b_1, t_{n+1} \rangle) : v, EN_{n+1}(\langle b_2, t_{n+1} \rangle) : v \rangle)$.

The observation that in D_{n+1} places b_1 and b_2 get the same tokens as a gets in D_n completes the proof of Lemma 17.

The correspondence of runs in this case is a bijection. Transitions are fired in the same order. The markings are equal to their counterparts in all the place that appear in both of the dataflows. Whereas a contains the same tokens as b_1 and b_2, which have to have identical content because, each of the transitions consuming token from one of those places consumes a token with identical history from the other one ($b_1\bullet = b_2\bullet$) and from Lemma 17 we know that there is no choice of such tokens, so it must be exactly the one consumed from the first place. The rest of the proof follows. □

7. The bioinformatics dataflow example revisited

We conjecture that in terms of expressible functions hierarchical dataflows are equivalent to NRC and thus, by following our claim in [20], are sufficient to describe most data-centric experiments in life sciences such as bioinformatics. To illustrate this we consider again the dataflow in Fig. 7. Closer inspection of this dataflow shows that it is not hierarchical. This is because the iterations in the dataflow start with a transition that only has unnesting edges as outgoing edges. This is in conflict with the *iteration split* rule in Fig. 9 which requires that next to the unnest-nest branch there is another branch that does not unnest and nest. Recall that the reason for this requirement is that if the function associated with the initial transition produces the empty set then the transition produces no tokens and the workflow will probably not terminate properly. Observe that this is indeed what happens if the workflow is presented with an empty "healthy" or "diseased" peptide list since the ∪ transition will never be enabled. The dataflow is therefore strictly speaking not semi-sound and cannot deal correctly with all possible input values. This soundness problem can be easily solved by introducing extra branches for the synchronization of the iterations as is shown in Fig. 13.

The corrected version of the dataflow can be shown to be hierarchical, which is demonstrated in Fig. 14 where the corresponding blank dataflow, called BDF_8 here, is generated from the blank dataflow with only one place, called BDF_1. The gray boxes indicate groups of nodes that were generated by expanding a single node in the preceding blank dataflow. For example, all nodes in BDF_2 where generated from the place in BDF_1 by applying the *sequential place split* and *sequential transition split*. For BDF_3 a place is split by using the *AND-split* and a transition is split by applying *iteration split*. In the following step BDF_4 is generated by applying the *sequential place split* to two places. Then BDF_5 is generated by using the *AND-split* for two places. Then for constructing BDF_6 some of the places that were just introduced are expanded with the *sequential place split*. In the next step BDF_7 is constructed by applying the *iteration split* to four transitions and the *AND-split* to two places. Finally, to construct

Author's personal copy

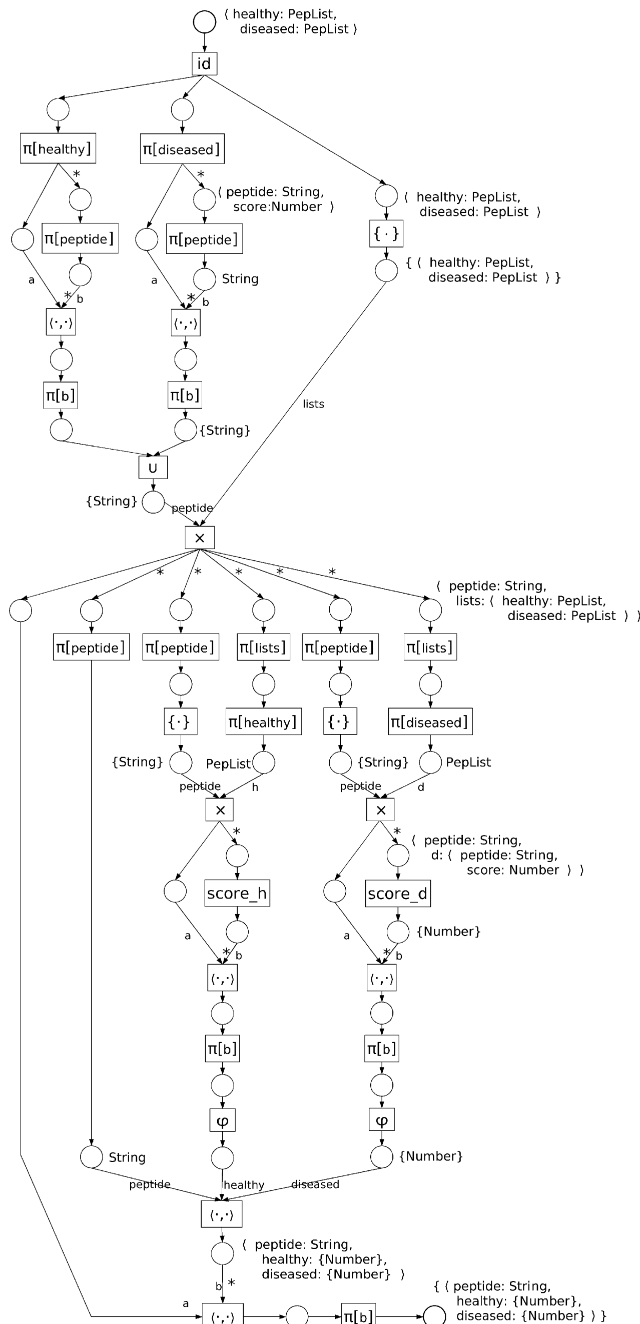


Fig. 13. Finding differences in peptide content of two samples (hierarchical).

BDF_7 the *sequential place split* and *sequential transition split* are applied several times.

As the preceding example shows, the hierarchical analysis of a dataflow can sometimes reveal subtle soundness problems. We intend to create a tool in which it would not only be possible to construct dataflows by refinement, but also arbitrarily constructed dataflows could be tested if they are hierarchical. For the rules proposed by Chrzastowski-Wachtel et al. for plain Petri nets a similar test is possible in polynomial time [21].

8. Conclusions and further research

In this paper we have presented DFL—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 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 semi-sound dataflows.

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 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 *run* of a dataflow 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 run of a dataflow 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 find for example similar dataflows or dataflows that can be reused for the current research problem. Since dataflows in our language are essentially labeled graphs it seems likely that a suitable existing graph-based query formalism could be employed 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

Author's personal copy

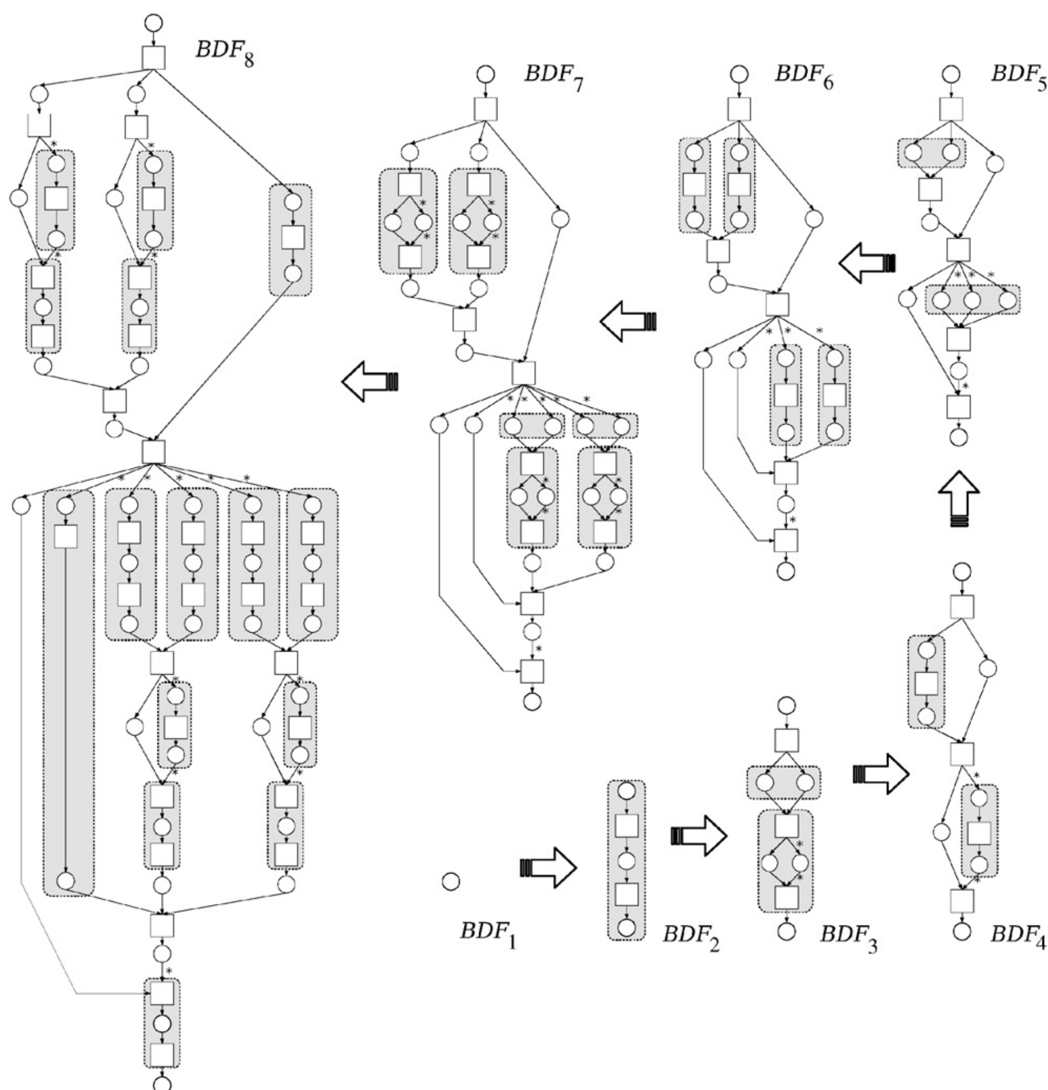


Fig. 14. The generation of the blank dataflow from Fig. 7.

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] W.M.P. van der Aalst, The application of petri nets to workflow management, J. Circuits Syst. Comput. 8 (1) (1998) 21–66.

[2] K. Jensen, Coloured Petri Nets Basic Concepts Analysis Methods and Practical Use. vols. 1 and 2, second ed., Springer, London, UK, 1996.

[3] R. Valk, Object Petri nets: using the nets-within-nets paradigm, in: Lectures on Concurrency and Petri Nets, 2003, pp. 819–848.

[4] R. Valk, Self-modifying nets a natural extension of Petri nets, in: ICALP, 1978, pp. 464–476.

[5] A. Oberweis, P. Sander, Information system behavior specification by high level petri nets, ACM Trans. Inf. Syst. 14 (4) (1996) 380–420.

[6] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, J. Van den Bussche, Petri net+nested relational calculus=dataflow, in: OTM Conferences (1), 2005, pp. 220–237.

[7] P. Buneman, S. Naqvi, V. Tannen, L. Wong, Principles of programming with complex objects and collection types, Theor. Comput. Sci. 149 (1) (1995) 3–48.

[8] T. Murata, Petri nets: properties analysis and applications, Proc. IEEE 77 (4) (1989) 541–580.

[9] W. Reisig, Petri Nets: An Introduction, Springer, New York, Inc., NY, USA, 1985.

[10] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Greenwood, T. Carver, A. Wipat, P. Li, Taverna: a tool for the composition and enactment of bioinformatics workflows, Bioinformatics 20 (17) (2004) 3045–3054.

[11] Object Management Group, Unified modeling language resource page, ⟨http://www.uml.org/⟩.

Author's personal copy

[12] S. Altschul, W. Gish, W. Miller, E. Myers, D. Lipman, Basic local alignment search tool, J. Mol. Biol. 215 (3) (1990) 403–410.

[13] B. Boeckmann, A. Bairoch, R. Apweiler, MC. Blatter, A. Estreicher, et al., The swiss-prot protein knowledgebase and its supplement trembl in 2003, Nucleic Acids Res. 31 (2003) 365–370.

[14] P. Rice, I. Longden, A. Bleasby, Emboss: the European molecular biology open software suite (2000), Trends Genet. 16 (6) (2000) 276–277.

[15] D. Dumont, J.P. Noben, J. Raus, P. Stinissen, J. Robben, Proteomic analysis of cerebrospinal fluid from multiple sclerosis patients, Proteomics 4 (7) (2004).

[16] G. Berthelot, Checking properties of nets using transformation, in: Advances in Petri Nets 1985, Covers the 6th European Workshop on Applications and Theory in Petri Nets-selected Papers, Lecture Notes in Computer Science, vol. 222. London, UK, Springer, Berlin, 1986, pp. 19–40.

[17] W.M.P. van der Aalst, Verification of workflow nets, in: ICATPN '97: Proceedings of the 18th International Conference on Application and Theory of Petri Nets, London, UK, Springer, Berlin, 1997, pp. 407–426

[18] H.A. Reijers, Design and Control of Workflow Processes: Business Process Management for the Service Industry, Lecture Notes in Computer Science, vol. 2617, Springer, New York, Inc., Secaucus, NJ, USA, 2003.

[19] P. Chrzastowski-Wachtel, B. Benatallah, R. Hamadi, M. O'Dell, A. Susanto, A top-down petri net-based approach for dynamic workflow modeling, in: W.M.P. van der Aalst, A.H.M. ter Hofstede, M. Weske (Eds.), Business Process Management, Lecture Notes in Computer Science, vol. 2678, Springer, Berlin, 2003 pp. 336–353.

[20] A. Gambin, J. Hidders, N. Kwasnikowska, S. Lasota, J. Sroka, J. Tyszkiewicz, J. Van den Bussche, NRC as a formal model for expressing bioinformatics workflows, Poster at ISMB, 2005.

[21] P. Chrzastowski-Wachtel, private communication, 2007.