Formalized Data Flow Diagrams and Their Relation to Other Computational Models

Jürgen Symanzik and Albert L. Baker

TR #96–20

December 1996

Keywords: Computational Power, Turing Machine, FIFO Petri Net, Program Machine.

Department of Computer Science

226 Atanasoff Hall

Iowa State University

Ames, Iowa 50011–1040, USA

1 FORMALIZED DATA FLOW DIAGRAMS AND THEIR RELATION TO OTHER COMPUTATIONAL MODELS

Abstract

One approach to the formalization of Data Flow Diagrams (DFD's) is presented by Coleman ([Col91], [CB94]) and Leavens, et al., [LWBL96]. These Formalized Data Flow Diagrams (FDFD's) can be viewed as another model of computation. This paper contains an analysis of the computational power of these FDFD's. We first consider the issue whether certain features of FDFD's affect their computational power. A Reduced Data Flow Diagram (RDFD) is an FDFD with no stores, finite domains for flow values, and no facility for testing for empty flows, but it may contain persistent flows. An RDFD without persistent flows is called a persistent flow–free Reduced Data Flow Diagram (PFF–RDFD). We show that PFF–RDFD's are Turing equivalent. The other features of FDFD's only add to the expressive power of FDFD's ([SB96]). Therefore, any FDFD can be expressed as an PFF–RDFD. Our proof that PFF–RDFD's are Turing equivalent procedes as follows. We first show that any RDFD can be simulated by a FIFO Petri Net. We then show that any Program Machine can be simulated by an PFF–RDFD. It is known that FIFO Petri Nets and Program Machines both are Turing equivalent.

1.1 Introduction

Traditional Data Flow Diagrams (DFD's) are the cornerstone of the software development methodology known as "Structured Analysis" (SA) ([DeM78], [WM85a]), and they are probably the most widely used specification technique in industry today ([BB93]). DFD's are popular because of their graphical representation and their hierarchical structure. Thus, they are ideally suited for users with non–technical backgrounds and are commonly used to depict the static structure of information flow in a system.

Traditional DFD's consist of a set of bubbles and a set of labeled flows. Bubbles represent either processes in a concurrent system or sequential procedures and are usually drawn as circles, ovals, or boxes. Flows represent data paths and are drawn as arrows connecting the bubbles. Flows coming into a bubble are called inflows and flows leaving a bubble are called outflows. Informally, a bubble reads the information on its inflows, and produces new information on its outflows. There are two types of flows. Persistent flows are like shared variables whose values are written by the source bubble and read by the destination bubble. Consumable flows are modelled as unbounded FIFO queues, with the source bubble enqueueing values on the queue and the destination bubble dequeueing values.

Even though traditional DFD's are popular, they lack formality and do not provide a rigorous definition of system functionality. Numerous attempts to formalize DFD's have appeared in the technical literature, e. g., in [DeM78], [WM85a], [WM85b], [Har87], [TP89], [You89], [Har92], and [Har96]. We focus on the Formalized Data Flow Diagrams (FDFD's) developed by Coleman, Wahls, Baker, and Leavens in [Col91], [CB94], [WBL93], and [LWBL96]. We answer the following questions about the computational power of FDFD's. Do FDFD's have the same computational power as FIFO Petri Nets, another specification technique for concurrent and distributed systems? If so, to which class of FIFO Petri Nets are FDFD's equivalent? Are FDFD's Turing equivalent?

After definitions of FDFD's, Reduced Data Flow Diagrams (RDFD's), persistent flow–free RDFD's (PFF–RDFD's), FIFO Petri Nets, and Program Machines are given in Section 1.2, we answer the above questions in Section 1.3. Indeed, we will show that PFF–RDFD's and Turing Machines are equivalent. Actually, we will not directly compare PFF–RDFD's and Turing Machines, but instead provide simulations of RDFD's by FIFO Petri Nets and simulations of Program Machines by PFF–RDFD's. A summary and a preview of future work concludes this paper in Section 1.4.

1.2 Computational Models

In this section we define FDFD's, RDFD's, and PFF–RDFD's, summarize the definitions of FIFO Petri Nets and Program Machines, and introduce the overall concept of a Computation System.

1.2.1 FDFD's and RDFD's

We will use the formal definitions from [LWBL96] to introduce Formalized Data Flow Diagrams (FDFD's). The cited paper contains a more detailed explanation of the underlying operational semantics of FDFD's and an extended example.

In all our definitions, we use the notation $X_\perp$ for the set $X \cup \{\perp\}$, where $\perp$ means no information.

Definition (1.2.1.1): A *Formalized Data Flow Diagram* (FDFD) is a quintuple

$$FDFD = (B, FLOWNAMES, TYPES, P, F),$$

where B is a set of *bubbles*, $FLOWNAMES$ is a set of *flows*, $TYPES$ is a set of *types*, P is the set $\{persistent, consumable\}$ and $F = B \times FLOWNAMES \times TYPES \times B \times P$. The following notational convention for members from these domains is used: $b \in B, fn \in FLOWNAMES, T \in TYPES, p \in P, f \in F$. ■

Definition (1.2.1.2): We define the following auxiliary functions for describing the structure of an FDFD:

Function : Type
$Source : F \to B$
$FlowName : F \to FLOWNAMES$
$TypeOf : F \to TYPES$
$Target : F \to B$
$Consumable : F \to Boolean$
$Inputs : B \to PowerSet(F)$
$Outputs : B \to PowerSet(F)$
$TypeMeaning : TYPES \to Set$

■

In the previous definition, we think of a type as describing a set of objects. The set of objects associated with a type T is given by $TypeMeaning(T)$. By Set, we mean the class of all recursive sets.

Definition (1.2.1.3): We define the following domains describing the configuration of an FDFD:

Notation for Members	$\in$ Name	$=$ description
m	$\in MODES$	$= \{idle,\ working\}$
bm	$\in BubbleMode$	$= B \to MODES$
o	$\in OBJECTS$	$= \bigcup_{T \in TYPES} TypeMeaning(T)$
s	$\in WhatRead$	$= (F \to OBJECTS_\perp)$
r	$\in Read$	$= B \to WhatRead$
fs	$\in FlowState$	$= F \to OBJECTS^*$
$\gamma = (bm, r, fs)$	$\in$,	$= BubbleMode \times Read \times FlowState$

We call $\gamma = (bm, r, fs) \in$, a *state* (or a *configuration*) of an FDFD. ■

Definition (1.2.1.4): Sequences of objects, i. e., elements of $(OBJECTS^*)_\perp$, are treated as FIFO queues using the following constants and operations:

$$
\begin{aligned}
\langle\rangle &: OBJECTS^* \\
Enq &: (OBJECTS^*)_\perp \times OBJECTS \to (OBJECTS^*)_\perp \\
IsEmpty &: (OBJECTS^*)_\perp \to Boolean_\perp \\
Head &: (OBJECTS^*)_\perp \to OBJECTS_\perp \\
Rest &: (OBJECTS^*)_\perp \to (OBJECTS^*)_\perp
\end{aligned}
$$

These operations are defined to satisfy the following equations for all $q \in OBJECTS^*$ and $o \in OBJECTS$.

$$
\begin{aligned}
Enq(\perp, o) &= \perp \\
IsEmpty(\perp) &= \perp \\
IsEmpty(\langle\rangle) &= true \\
IsEmpty(Enq(q, o)) &= false \\
Head(\perp) &= \perp \\
Head(\langle\rangle) &= \perp \\
Head(Enq(q, o)) &= \textbf{if } IsEmpty(q) \textbf{ then } o \textbf{ else } Head(q) \textbf{ fi} \\
Rest(\perp) &= \perp \\
Rest(\langle\rangle) &= \perp \\
Rest(Enq(q, o)) &= \textbf{if } IsEmpty(q) \textbf{ then } \langle\rangle \textbf{ else } Enq(Rest(q), o) \textbf{ fi}
\end{aligned}
$$

■

The things that the bubbles in an FDFD can do are defined through three curried functions *Enabled*, *Consume*, and *Produce* defined as follows:

Definition (1.2.1.5): Enablement of a bubble can depend on both the presence of values on input flows as well as on the values on such flows:

$$Enabled : B \to (FlowState \to Boolean_{\perp})$$

A bubble that is enabled can change its mode from *idle* to *working*. It reads some of its inflows, and consumes some of these. Only consumable flows may be consumed and consumption means removing the head of the sequence associated with the flow:

$$Consume : B \to ((FlowState \times Read) \to PowerSet(FlowState \times Read)).$$

A bubble in *working* mode can produce some output in the transition from *working* to *idle*:

$$Produce : B \to ((FlowState \times Read) \to PowerSet(FlowState \times Read)).$$

The notation $[x \mapsto y]g$ is an update to a function, g, and it is defined by the following equation.

$$[x \mapsto y]g \stackrel{\text{def}}{=} \lambda z \,.\, \textbf{if } z = x \textbf{ then } y \textbf{ else } g(z) \textbf{ fi}$$

The curried function $In(f, b)$ represents the changes that b makes to the flow state and read function by reading the flow f, and it is defined as follows:

$In : (F \times B) \to ((FlowState \times Read) \to (FlowState \times Read))$

$In(f, b)(fs, r) =$

$\quad$ **let** $r_b = [f \mapsto Head(fs(f))](r(b))$

$\quad$ **in** (**if** $Consumable(f)$ **then** $[f \mapsto Rest(fs(f))]fs$ **else** fs **fi**,

$\quad\quad [b \mapsto r_b]r)$

By analogy, the curried function $Out(o, f, b)$ represents the changes that b makes to the flow state and read function by producing o on the flow f, and it is defined as follows:

$Out : (OBJECTS \times F \times B) \to ((FlowState \times Read) \to (FlowState \times Read))$

$Out(o, f, b)(fs, r) =$

$\quad$ **let** $r_b = \lambda f' \,.\, \perp$

$\quad$ **in** (**if** $Consumable(f)$ **then** $[f \mapsto Enq(fs(f), o)]fs$ **else** $[f \mapsto Enq(\langle\rangle, o)]fs$ **fi**,

$\quad\quad [b \mapsto r_b]r)$ ■

Two kinds of transitions are allowed between configurations: an enabled bubble can go from *idle* to *working*, and a *working* bubble can go to *idle*. We use the symbol $\longrightarrow$ to state the binary relation between configurations.

The transition rule

$$\frac{\begin{array}{c} bm(b) = idle, \\ Enabled(b)(fs) = true, \\ bm' = [b \mapsto working]bm, \\ (fs', r') \in Consume(b)(fs, r) \end{array}}{(bm, r, fs) \longrightarrow (bm', r', fs')}$$

states that if b is *idle* and enabled, then it may change its mode to *working* and consume some of its inputs. Finally, if the conditions above the horizontal line hold, then the transition below the line may take place.

The transition rule

$$\frac{\begin{array}{c} bm(b) = working, \\ bm' = [b \mapsto idle]bm, \\ (fs', r') \in Produce(b)(fs, r) \end{array}}{(bm, r, fs) \longrightarrow (bm', r', fs')}$$

states that if b is *working*, then it may change its mode to *idle* and produce some outputs.

We next define what we mean by a Reduced Data Flow Diagram (RDFD). We do not explicitely state that there are no stores in RDFD's since stores are considered as an addition to, but not as a part of the FDFD's defined in [LWBL96].

Definition (1.2.1.6): An FDFD that obeys to the following restrictions is called a *Reduced Data Flow Diagram* (RDFD):

(i) The set $OBJECTS$, that contains all types of objects that may appear on flows, is finite.

(ii) The mappings *Enabled*, *Consume*, and *Produce* contain no higher logical constructor than First Order Predicate Calculus (FOPC) assertions over the values on the inflows and outflows of the bubble. Each assertion must be computable ([WBL93]).

(iii) The mappings $Enabled(b)$ and $Consume(b)$ do not make use of the positive form of the operation *IsEmpty*. Instead, it only occurs with a negation, i. e., $\neg IsEmpty(fs(f))$, where $f \in Inputs(b)$.

(iv) The mappings $Enabled(b)$ and $Consume(b)$ only make use of the *Head* element of a flow $f \in Inputs(b)$. Constructions such as "**if** $Head(Rest(fs(f))) = x$ **then** ... **fi**" are not allowed.

(v) Every flow is initialized, i. e., $\forall\ f \in F\ :\ fs_{initial}(f) \in (TypeOf(f))^*$. Also, $bm_{initial} = \lambda b\ .\ idle$ and $r_{initial} = \lambda b\ .\ \lambda f\ .\ \perp$. ■

The finiteness of $OBJECTS = \bigcup_{T \in TYPES} TypeMeaning(T)$ claimed in (i) guarantees that each simple primitive type and each structured primitive type only has a fixed number of elements. E. g., we can enumerate $TypeMeaning(INTEGER) = \{MININT, \ldots, MAXINT\}$ and assuming that T satisfies this criterion, then

$$TypeMeaning(Sequence\ of\ T) = \{s \mid length(s) \leq c, \forall\, k \leq c\ :\ s[k] \in T\}$$

for some constant c.

By (ii), we can determine the result of each expression for every possible combination of inputs for this expression. Especially, we can reformulate every expression in the mappings *Enabled*, *Consume*, and *Produce* as finite many **if − then − fi**–statements that cover possible combinations of inputs. However, these statements may be nondeterministic.

Therefore, we can rewrite the mappings *Enabled*, *Consume*, and *Produce* of any RDFD in the following normal form. Later, we will use this normal form to simulate RDFD's by FIFO Petri Nets. If not otherwise stated, we use this normal form for all examples throughout this paper.

Definition (1.2.1.7): The *normal form* of an RDFD (*nf–RDFD*) is defined as follows:

$B_{FDFD} = \{b_1, \ldots, b_b\}$

$FLOWNAMES_{FDFD} = \{f_1, \ldots, f_f\}$

A bubble b_i can be never enabled (case E_0 below), it can be always enabled (case E_1), or it is enabled if at least one of its m_i enabling conditions of its enabling rule yields *true* (case E_2). An enabling condition j will yield *true* if the n_{ij1} consumable inflows $f^c_{ij1}, \ldots, f^c_{ijn_{ij1}}$ are not empty, the head elements of these inflows are the values $o^c_{ij1}, \ldots, o^c_{ijn_{ij1}}$, and the data on the n_{ij2} persistent inflows $f^p_{ij1}, \ldots, f^p_{ijn_{ij2}}$ are the values $o^p_{ij1}, \ldots, o^p_{ijn_{ij2}}$. An enabling condition may access only consumable flows (then $n_{ij2} = 0$), only persistent flows (then $n_{ij1} = 0$), or consumable and persistent flows (then $n_{ij1} > 0$ and $n_{ij2} > 0$). Also, f^c_{ijk} and f^c_{ilm} as well as f^p_{ijk} and f^p_{ilm} may be identical for some combination of (j,k) and (l,m) pairs.

$\forall\, b_i \in B\ :$

$$Enabled(b_i) = \lambda fs\ .\ false \qquad (E_0)$$

or

$$Enabled(b_i) = \lambda fs\ .\ \ true \qquad (E_1)$$

or

$$Enabled(b_i) = \lambda fs\ . \qquad (E_2)$$

$$\begin{aligned}
&(\neg IsEmpty(fs(f^c_{i11})) \wedge \ldots \wedge \neg IsEmpty(fs(f^c_{i1n_{i11}}))\\
&\qquad \wedge Head(fs(f^c_{i11})) = o^c_{i11} \wedge \ldots \wedge Head(fs(f^c_{i1n_{i11}})) = o^c_{i1n_{i11}}\\
&\qquad \wedge Head(fs(f^p_{i11})) = o^p_{i11} \wedge \ldots \wedge Head(fs(f^p_{i1n_{i12}})) = o^p_{i1n_{i12}})\\
&\vee \ldots \vee\\
&(\neg IsEmpty(fs(f^c_{im_i1})) \wedge \ldots \wedge \neg IsEmpty(fs(f^c_{im_in_{im_i1}}))\\
&\qquad \wedge Head(fs(f^c_{im_i1})) = o^c_{im_i1} \wedge \ldots \wedge Head(fs(f^c_{im_in_{im_i1}})) = o^c_{im_in_{im_i1}}\\
&\qquad \wedge Head(fs(f^p_{im_i1})) = o^p_{im_i1} \wedge \ldots \wedge Head(fs(f^p_{im_in_{im_i2}})) = o^p_{im_in_{im_i2}})
\end{aligned}$$

where $\forall\, i \in \{1, \ldots, b\}$:

$$m_i \geq 1,\ n_{i11}, n_{i12}, \ldots, n_{im_i1}, n_{im_i2} \geq 0$$

$$F^c_i = \{f^c_{i11}, \ldots, f^c_{i1n_{i11}}, \ldots, f^c_{im_i1}, \ldots, f^c_{im_in_{im_i1}}\}$$

$$F^p_i = \{f^p_{i11}, \ldots, f^p_{i1n_{i12}}, \ldots, f^p_{im_i1}, \ldots, f^p_{im_in_{im_i2}}\}$$

$$F_i = F^c_i \cup F^p_i = Inputs(b_i)$$

$$\forall\, f^c \in F^c_i\ :\ Consumable(f^c)$$

$$\forall\, f^p \in F^p_i\ :\ \neg Consumable(f^p)$$

$$O^c_i = \{o^c_{i11}, \ldots, o^c_{i1n_{i11}}, \ldots, o^c_{im_i1}, \ldots, o^c_{im_in_{im_i1}}\}$$

$$O^p_i = \{o^p_{i11}, \ldots, o^p_{i1n_{i12}}, \ldots, o^p_{im_i1}, \ldots, o^p_{im_in_{im_i2}}\}$$

$$O_i = O^c_i \cup O^p_i$$

$$\bigcup_{i=1,\ldots,b} F_i \subset F_{FDFD}$$

$$\bigcup_{i=1,\ldots,b} O_i \subset OBJECTS$$

A bubble b_i may consume nothing if it is never enabled or always enabled (case C_1 below). Otherwise, if several of the m_i enabling conditions of its enabling rule are $true$, b_i nondeterministically selects one of these $Consume$ cases, say j, and it consumes from the n_{ij1} consumable inflows $f^c_{ij1}, \ldots, f^c_{ijn_{ij1}}$ and the n_{ij2} persistent inflows $f^p_{ij1}, \ldots, f^p_{ijn_{ij2}}$ of b_i referred to in this $Consume$ case (case C_2). This implies that the head elements of the consumable inflows are removed from their queue. The persistent inflows are not modified at all.

$\forall\, b_i \in B$:

$$Consume(b_i) = \lambda(fs, r)\ .\ \{(fs, r)\} \qquad (C_1)$$

or

$$Consume(b_i) = \lambda(fs, r)\ . \qquad (C_2)$$

$\{$**if** $\neg IsEmpty(fs(f^c_{i11})) \wedge \ldots \wedge \neg IsEmpty(fs(f^c_{i1n_{i11}}))$

$\wedge Head(fs(f^c_{i11})) = o^c_{i11} \wedge \ldots \wedge Head(fs(f^c_{i1n_{i11}})) = o^c_{i1n_{i11}}$

$\wedge Head(fs(f^p_{i11})) = o^p_{i11} \wedge \ldots \wedge Head(fs(f^p_{i1n_{i12}})) = o^p_{i1n_{i12}})$

then $In(f^c_{i11}, b_i)(\ldots(In(f^c_{i1n_{i11}}, b_i)(In(f^p_{i11}, b_i)(\ldots(In(f^p_{i1n_{i12}}, b_i)(fs, r))\ldots)))\ldots)$

fi,

$\ldots$,

if $\neg IsEmpty(fs(f^c_{im_i1})) \wedge \ldots \wedge \neg IsEmpty(fs(f^c_{im_in_{im_i1}}))$

$\wedge Head(fs(f^c_{im_i1})) = o^c_{im_i1} \wedge \ldots \wedge Head(fs(f^c_{im_in_{im_i1}})) = o^c_{im_in_{im_i1}}$

$\wedge Head(fs(f^p_{im_i1})) = o^p_{im_i1} \wedge \ldots \wedge Head(fs(f^p_{im_in_{im_i2}})) = o^p_{im_in_{im_i2}})$

then $In(f^c_{im_i1}, b_i)(\ldots(In(f^c_{im_in_{im_i1}}, b_i)(In(f^p_{im_i1}, b_i)(\ldots(In(f^p_{im_in_{im_i2}}, b_i)(fs, r))\ldots)))\ldots)$

fi

$\}$

A bubble b_i may produce nothing and simply reset its internal status independently from what it has read (case P_0 below). It may nondeterministically select one of its s_i *Produce* cases, say j, producing objects $u_{ij1}, \ldots, u_{ijk_{ij}}$ on its outflows $h_{ij1}, \ldots, h_{ijk_{ij}}$ (case P_1). Otherwise, if it has read some input that matches one of the m_i *Consume* cases, say j, it nondeterministically selects one of the l_{ij} *Produce* subcases, say k, and produces objects $u_{ijk1}, \ldots, u_{ijkq_{ijk}}$ on its outflows $h_{ijk1}, \ldots, h_{ijkq_{ijk}}$ (case P_2). We use the symbol $\Box$ to indicate that any of the l_{ij} *Produce* subcases can be selected nondeterministically if *Produce* case j has been selected (nondeterministically).

$\forall\, b_i \in B$:

$$Produce(b_i) = \lambda(fs, r)\ .\ \{(fs, [b_i \mapsto \lambda f\ .\ \bot]r)\} \qquad (P_0)$$

or

$$Produce(b_i) = \lambda(fs, r)\ . \qquad (P_1)$$

$\{\, Out(u_{i11}, h_{i11}, b_i)(\ldots(Out(u_{i1k_{i1}}, h_{i1k_{i1}}, b_i)(fs, r))\ldots),$

$\ldots$,

$Out(u_{is_i1}, h_{is_i1}, b_i)(\ldots(Out(u_{is_ik_{is_i}}, h_{is_ik_{is_i}}, b_i)(fs, r))\ldots)\}$

$\}$

or

$$Produce(b_i) = \lambda(fs, r)\ . \qquad (P_2)$$

$\{$**if** $r(b_i)(f^c_{i11}) = o^c_{i11} \wedge \ldots \wedge r(b_i)(f^c_{i1n_{i11}}) = o^c_{i1n_{i11}}$

$\wedge r(b_i)(f^p_{i11}) = o^p_{i11} \wedge \ldots \wedge r(b_i)(f^p_{i1n_{i12}}) = o^p_{i1n_{i12}}$

$$
\begin{array}{l}
\textbf{then } Out(u_{i111}, h_{i111}, b_i)(\ldots(Out(u_{i11q_{i11}}, h_{i11q_{i11}}, b_i)(fs, r))\ldots) \\
\qquad \Box \ldots \\
\qquad \vdots \\
\qquad \Box\, Out(u_{i1l_{i1}1}, h_{i1l_{i1}1}, b_i)(\ldots(Out(u_{i1l_{i1}q_{i1l_{i1}}}, h_{i1l_{i1}q_{i1l_{i1}}}, b_i)(fs, r))\ldots) \\
\textbf{fi}, \\
\ldots, \\
\textbf{if } r(b_i)(f^c_{im_i1}) = o^c_{im_i1} \wedge \ldots \wedge r(b_i)(f^c_{im_in_{im_i1}}) = o^c_{im_in_{im_i1}} \\
\qquad \wedge r(b_i)(f^p_{im_i1}) = o^p_{im_i1} \wedge \ldots \wedge r(b_i)(f^p_{im_in_{im_i2}}) = o^p_{im_in_{im_i2}} \\
\textbf{then } Out(u_{im_i11}, h_{im_i11}, b_i)(\ldots(Out(u_{im_i1q_{im_i1}}, h_{im_i1q_{im_i1}}, b_i)(fs, r))\ldots) \\
\qquad \Box \ldots \\
\qquad \vdots \\
\qquad \Box\, Out(u_{im_il_{im_i}1}, h_{im_il_{im_i}1}, b_i)(\ldots(Out(u_{im_il_{im_i}q_{im_il_{im_i}}}, h_{im_il_{im_i}q_{im_il_{im_i}}}, b_i)(fs, r))\ldots) \\
\textbf{fi} \\
\}
\end{array}
$$

where $\forall\, i \in \{1, \ldots, b\}$:

$$
\begin{array}{l}
m_i \geq 1,\ l_{i1}, \ldots, l_{im_i}, q_{i11}, \ldots, q_{im_il_{im_i}} \geq 0 \\
H_i = \{h_{i111}, \ldots, h_{i11q_{i11}}, \ldots, h_{im_il_{im_i}1}, \ldots, h_{im_il_{im_i}q_{im_il_{im_i}}}\} \\
U_i = \{u_{i111}, \ldots, u_{i11q_{i11}}, \ldots, u_{im_il_{im_i}1}, \ldots, u_{im_il_{im_i}q_{im_il_{im_i}}}\} \\
\bigcup_{i=1,\ldots,b} H_i \subset F_{FDFD} \\
\bigcup_{i=1,\ldots,b} U_i \subset OBJECTS
\end{array}
$$

■

We assume $Consume(b_i)$ and $Produce(b_i)$ are finite enumerable and ordered sets for every $b_i \in B$. Thus, we can access the jth element of $Consume(b_i)$ through $(Consume(b_i))_j$ and the kth element of $Produce(b_i)$ through $(Produce(b_i))_k$.

Definition (1.2.1.8): An RDFD that does not have any persistent flows is called a *persistent flow–free Reduced Data Flow Diagram* (PFF–RDFD). This implies that $F = B \times FLOWNAMES \times TYPES \times B \times \{consumable\}$. ■

Definition (1.2.1.9): A *firing sequence* (*computation sequence*) of an FDFD is a possibly infinite sequence $(b_i, a_i, j_i) \in B \times \{C, P\} \times I\!N, i \geq 0$, such that, if transition (b_i, a_i, j_i) is fired in state (bm, r, fs), then

$$(fs', r') = \begin{cases} (Consume(b_i))_{j_i}(fs, r), & \text{if } a_i = C \\ (Produce(b_i))_{j_i}(fs, r), & \text{if } a_i = P \end{cases}$$

$$bm'(b_i) = \begin{cases} working, & \text{if } a_i = C \\ idle, & \text{if } a_i = P \end{cases}$$

$$bm'(b) = bm(b) \quad \forall\, b \in B - \{b_i\}$$

and

$$(bm, r, fs) \rightarrow (bm', r', fs').$$

We introduce the notation $(bm, r, fs)[(b, a, j)]$ to indicate that transition (b, a, j) is fireable in state (bm, r, fs) and $(bm, r, fs)[(b, a, j)](bm', r', fs')$ to indicate that state (bm', r', fs') is reached upon the firing of transition (b, a, j) in state (bm, r, fs).

By induction, we extend this notation for firing sequences:

$$(bm_0, r_0, fs_0)[(b_1, a_1, j_1), \ldots, (b_{n-1}, a_{n-1}, j_{n-1}), (b_n, a_n, j_n)]$$

is used to indicate that transition (b_n, a_n, j_n) is fireable in state $(bm_{n-1}, r_{n-1}, fs_{n-1})$, given that

$$(bm_0, r_0, fs_0)[(b_1, a_1, j_1), \ldots, (b_{n-1}, a_{n-1}, j_{n-1})](bm_{n-1}, r_{n-1}, fs_{n-1})$$

holds. By analogy, we use

$$(bm_0, r_0, fs_0)[(b_1, a_1, j_1), \ldots, (b_n, a_n, j_n)](bm_n, r_n, fs_n)$$

to indicate that state (bm_n, r_n, fs_n) is reached upon the firing of the sequence $(b_1, a_1, j_1), \ldots, (b_n, a_n, j_n)$.

■

Definition (1.2.1.10): The *set of firing sequences* (*set of computation sequences, language*) of an FDFD, denoted by $FS(FDFD, \gamma_{initial})$, is the set containing all firing sequences that are possible for this FDFD, given $\gamma_{initial} = (bm_{initial}, r_{initial}, fs_{initial}) = (bm_0, r_0, fs_0)$, i. e.,

$$FS(FDFD, \gamma_{initial}) = \{s \mid s \in (B \times \{C, P\} \times I\!N)^* \wedge \gamma_{initial}[s]\}.$$ ■

Definition (1.2.1.11): The *Reachability Set* of an FDFD, denoted by $RS(FDFD, \gamma_{initial})$, is the set of states $\gamma = (bm, r, fs)$ that are reachable from $\gamma_{initial} = (bm_{initial}, r_{initial}, fs_{initial})$, i. e.,

$$RS(FDFD, \gamma_{initial}) = \{\gamma \mid \gamma \in , \ \wedge\, \exists\, s \in FS(FDFD, \gamma_{initial}) \ : \ \gamma_{initial}[s]\gamma\}.$$ ■

1.2.2 FIFO Petri Nets

In some sense, FIFO Petri Nets (introduced in [MM81]) are Petri Nets where places contain words instead of tokens and arcs are labelled by words. More formally, we make use of the definition of FIFO Petri Nets as given in [Rou87].

Definition (1.2.2.1): A *FIFO Petri Net* is a quintuple $FPN = (P, T, B, F, Q)$ where P is a finite set of *places* (also called *queues*), T is a finite set of *transitions* (disjoint from P), Q is a finite *queue alphabet*, and $F : T \times P \to Q^*$ and $B : P \times T \to Q^*$ are two mappings called respectively *forward* and *backward incidence mappings*. ■

Definition (1.2.2.2): A *marking* M of a FIFO Petri Net is a mapping $M : P \to Q^*$.

A transition t is *fireable* in M, written $M(t>$, if $\forall p \in P : B(p,t) \leq M(p)$ (where $u \leq x$ means u is a prefix of x).

For a marking M, we define the firing of a transition t, written $M(t > M'$, if $M(t >$ and the following equation between words holds $\forall p \in P : B(p,t)M'(p) = M(p)F(t,p)$. That means, the firing of a transition t removes $B(p,t)$ from the head of $M(p)$ and appends $F(t,p)$ to the end of the resulting word. ■

Definition (1.2.2.3): A FIFO Petri Net FPN together with an initial marking $M_0 : P \to Q^*$ is called a *marked FIFO Petri Net* and is denoted by (FPN, M_0).

As usual, the firing of a transition can be extended to the firing of a sequence of transitions. We denote by $FS(FPN, M_0)$ the *set of firing sequences* of this FIFO Petri Net. The firing of a sequence u of transitions from a marking M to a marking M' is written as $M(u > M'$.

The set of markings that are reachable from M_0 is called *Reachability Set* and it is denoted by $Acc(FPN, M_0)$. ■

1.2.3 Program Machines

We introduce Program Machines as given in [Min67], using the formalism of [VVN81].

Definition (1.2.3.1): A *Program Machine* is given by a finite set $R = \{r_1, \ldots, r_p\}$ of *registers*, a finite set $Q = \{q_0, \ldots, q_r\}$ of *labels*, and a finite set I of *instructions*. Each instruction is labelled by an element of Q and no two instructions have the same label. A Program Machine has exactly one "start

instruction"

$$q_0 \ : \textbf{start goto } q_1;$$

and exactly one "halt instruction"

$$q_f \ : \textbf{halt.}$$

The other instructions are of the type "increment register r_i"

$$q_s \ : r_i := r_i + 1 \textbf{ goto } q_m;$$

or "test and decrement register r_i"

$$q_s \ : \textbf{if } r_i = 0 \textbf{ then goto } q_l \textbf{ else } r_i := r_i - 1 \textbf{ goto } q_m.$$

The registers have nonnegative integers as values. First the start instruction is executed. Then the flow of control is determined by the goto contained in each instruction, until the halt instruction is reached.

■

Definition (1.2.3.2): A *computation sequence* of a Program Machine is a possibly infinite sequence

$$q_0, (k_1^1, \ldots, k_1^p, q_1), (k_2^1, \ldots, k_2^p, q_{i_2}), \ldots, (k_j^1, \ldots, k_j^p, q_{i_j}), \ldots, (k_r^1, \ldots, k_r^p, q_{i_r})$$

with $q_{i_r} = q_f$, i. e., the halt statement is reached. q_{i_j} is the actual instruction to be executed in step j and k_j^s is the content of register r_s just before the execution of instruction q_{i_j} in step j. In computations, arbitrary initial values $(k_1^1, \ldots, k_1^p) \in I\!N^p$ are allowed. Note that computations may be of infinite length.

■

1.2.4 Computation Systems and Homomorphisms

Now we will introduce the overall concept of a Computation System. All the definitions and results in this subsection are drawn from [KM82]. However, there exist similar approaches in the literature, e. g., in [Jen80], where the terms "transition system", "simulation", and "consistent homomorphism" have been defined with respect to a formal method that allows comparisons of the descriptive power of different types of Petri Nets.

For any set Σ, we denote by Σ^* the set of finite sequences of elements of Σ including the null sequence λ. The set of infinite sequences of elements of Σ is denoted by Σ^∞ and we define $\Sigma^\omega = \Sigma^\infty \cup \Sigma^*$.

Definition (1.2.4.1): A *computation system* $S = (\Sigma, D, x, \overline{\ \ })$ consists of a set D, an element x of D, a finite set Σ of *operations*, and a function "$\overline{\ \ }$" from Σ to the set of partial functions from D to

D. That is, for each $a \in \Sigma$, $\overline{a}$ is a partial function from D to D. The function "$\overline{\ }$" is extended to Σ^* by $\overline{\lambda} =$ identity, $\overline{\alpha\beta}(y) = \overline{\alpha} \cdot \overline{\beta}(y) = \overline{\beta}(\overline{\alpha}(y))$, $\alpha \cdot \beta \in \Sigma^*, y \in D$. ■

Here D is intuitively thought of as the set of states (or configurations) of the computation system, where a state includes control information as well as data for synchronization. The element x is then considered to be the initial state of the system. The performance of an operation will create a new state, as defined by the function $\overline{\ }$, and a sequence of operation performances can be thought of as a computation (or firing) sequence of the system.

Definition (1.2.4.2): Let $S = (\Sigma, D, x, \overline{\ })$ be a computation system. Let $y, z \in D$, and $\alpha \in \Sigma^*$. We define:

$$\begin{aligned}
\overline{\alpha}(y) &= \perp, && \text{if } \overline{\alpha}(y) \text{ is undefined} \\
\overline{\alpha}(y) &\neq \perp, && \text{if } \overline{\alpha}(y) \text{ is defined} \\
y &\xrightarrow{\alpha} z, && \text{if } \overline{\alpha}(y) = z \\
y &\xrightarrow{*} z, && \text{if } \exists\, \alpha \in \Sigma^* : y \xrightarrow{\alpha} z
\end{aligned}$$

■

Definition (1.2.4.3): Let $S = (\Sigma, D, x, \overline{\ })$ be a computation system. For each $y \in D$, a *computation sequence* from y is a (finite or infinite) sequence $a_1 a_2 \ldots$ of elements of Σ such that $\forall\, i : \overline{a_1 a_2 \ldots a_i}(y) \neq \perp$. We denote by $C_S^\omega(y)$ the *set of all computation sequences from* y and by $C_S(y) = \{\alpha \mid \alpha \in \Sigma^*, \overline{\alpha}(y) \neq \perp\}$ the *set of all finite computation sequences from* y. For each $y \in D$, we denote by $R_S(y) = \{\overline{\alpha}(y) \mid \alpha \in \Sigma^*, \overline{\alpha}(y) \neq \perp\}$ the *reachability set* from y. When $y = x$, we simply write R_S, C_S, and C_S^ω instead of $R_S(x)$, $C_S(x)$, and $C_S^\omega(x)$, respectively. ■

Note that for an $\alpha \in \Sigma^\omega$, α is in $C_S^\omega(y)$ if, and only if, every prefix of α is in $C_S(y)$.

The following definitions give some ideas about possible relations between different computation systems.

Definition (1.2.4.4): Let $S_1 = (\Sigma_1, D_1, x_1, \overline{\ }^1)$ and $S_2 = (\Sigma_2, D_2, x_2, \overline{\ }^2)$ be computation systems. A *homomorphism* $h = (\tau, \rho) : S_1 \to S_2$ consists of a homomorphism $\tau : \Sigma_1^* \to \Sigma_2^*$, and an injection $\rho : D_1 \to D_2$ which satisfies the following conditions:

$$\rho(x_1) = x_2 \qquad (1.2.4.4.1)$$

$$\forall\, y, z \in R_{S_1}\, \forall\, \alpha \in \Sigma_1^* : (y \xrightarrow{\alpha} z \Rightarrow \rho(y) \xrightarrow{\tau(\alpha)} \rho(z)) \qquad (1.2.4.4.2)$$

- A homomorphism is said to be *injective* if $\tau : \Sigma_1^* \rightarrow \Sigma_2^*$ is injective.

- A homomorphism is said to be *surjective* if the following conditions hold:

$$\forall\, y, z \in R_{S_1}\, \forall\, \beta \in \Sigma_2^* : (\rho(y) \xrightarrow{\beta} \rho(z) \Rightarrow \exists\, \alpha \in \Sigma_1^* : (\beta = \tau(\alpha) \wedge y \xrightarrow{\alpha} z)) \qquad (1.2.4.4.3)$$

$$\forall\, y \in R_{S_1}\, \forall\, z' \in D_2 : (\rho(y) \xrightarrow{*} z' \Rightarrow \exists\, z \in R_{S_1} : z' \xrightarrow{*} \rho(z)) \qquad (1.2.4.4.4)$$

- A homomorphism is said to be *bijective* if it is surjective and injective. ■

The injection ρ relates elements of D_1 to elements of D_2, where condition (1.2.4.4.1) states that the starting state of S_1 maps into the starting state of S_2. By condition (1.2.4.4.2), if α is any computation sequence in S_1 then there is a related computation sequence $\tau(\alpha)$ in S_2, with appropriate state mappings using ρ. Condition (1.2.4.4.3) means that any computation from $\rho(y)$ to $\rho(z)$ in S_2 is the image of a computation from y to z in S_1. Condition (1.2.4.4.4) means that any computation from $\rho(y)$ in S_2 is an initial segment of the image of some computation from y in S_1.

Definition (1.2.4.5): Let $S = (\Sigma, D, x, \overline{\ \ })$ be a computation system. Let $\alpha, \beta \in \Sigma^*$. We write $\alpha \simeq \beta$ if α is a permutation of β. Let $h = (\tau, \rho) : S_1 \rightarrow S_2$ be a homomorphism. h is said to be *spanning* if the following conditions hold:

$$\forall\, y, z \in R_{S_1}\, \forall\, \beta \in \Sigma_2^* : (\rho(y) \xrightarrow{\beta} \rho(z) \Rightarrow \exists\, \alpha \in \Sigma_1^* : \beta \simeq \tau(\alpha) \wedge y \xrightarrow{\alpha} z) \qquad (1.2.4.5.1)$$

$$\exists\, k \in I\!N\, \forall\, y \in R_{S_1}\, \forall\, y' \in D_2\, \forall\, \beta \in \Sigma_2^* : (\rho(y) \xrightarrow{\beta} y' \Rightarrow \exists\, \alpha \in \Sigma_1^*\, \exists\, z \in D_1\, \exists\, z' \in D_2\, \exists\, \beta', \beta'' \in \Sigma_2^* :$$

$$\rho(y) \xrightarrow{\tau(\alpha)} \rho(z) \xrightarrow{\beta''} z' \wedge y' \xrightarrow{\beta'} z' \wedge \tau(\alpha)\beta'' \simeq \beta\beta' \wedge \mid \beta'' \mid \leq k) \qquad (1.2.4.5.2)$$

$$(z \xrightarrow{\gamma} u \text{ in } S_1) \Rightarrow \exists\, u' \in D_2\, \exists\, \gamma', \gamma'' \in \Sigma_2^* : z' \xrightarrow{\gamma'} u' \wedge \rho(u) \xrightarrow{\gamma''} u' \wedge \tau(\gamma)\gamma'' \simeq \beta''\gamma' \qquad (1.2.4.5.3)$$

■

Condition (1.2.4.5.2) means that any computation β from $\rho(y)$ of S_2 is a prefix of a permutation of a computation of the form $\tau(\alpha)\beta'', \alpha \in \Sigma_1^*, \mid \beta'' \mid \leq k$. If $\alpha\gamma$ is a computation from y of S_1, then there must be a computation of the form $\beta\beta'\gamma'$ such that $\tau(\alpha)\tau(\gamma)$ is a prefix of a permutation of $\beta\beta'\gamma'$. Intuitively, β is an initial segment of a simulation of $\tau(\alpha)$, and if $\alpha\gamma$ is a computation of S_1, then β can be followed by a computation to simulate $\tau(\gamma)$.

Theorem (1.2.4.6): Every surjective homomorphism is a spanning homomorphism.

Definition (1.2.4.7): Let $h = (\tau, \rho) : S_1 \to S_2$ be a homomorphism. h is said to be *length preserving* if $\forall\, a \in \Sigma_1 : \mid \tau(a) \mid = 1$. ■

Definition (1.2.4.8): Let Σ be a finite set. For each $w \in \Sigma^*$, let $\iota(w)$ be the subset of Σ defined by $\iota(w) = \{a \mid \alpha a \beta = w,\, \alpha, \beta \in \Sigma^*,\, a \in \Sigma\}$.

Let $h = (\tau, \rho) : S_1 \to S_2$ be a surjective homomorphism. h is said to be *principal* if for each a and b in Σ_1, either $\tau(a) = \tau(b)$ or $\iota(\tau(a)) \cap \iota(\tau(b)) = \emptyset$, and $\overline{a} \neq \overline{b}$ implies $\iota(\tau(a)) \cap \iota(\tau(b)) = \emptyset$. ■

Theorem (1.2.4.9): Let $h = (\tau, \rho) : S_1 \to S_2$ be a bijective homomorphism. For each $y, z \in R_{S_1}$ and $\alpha \in \Sigma_1^*$ it holds that

$$y \xrightarrow{\alpha} z \text{ if, and only if, } \rho(y) \xrightarrow{\tau(\alpha)} \rho(z). \qquad (1.2.4.9.1)$$

Definition (1.2.4.10): Let $h = (\tau, \rho) : S_1 \to S_2$ be a homomorphism. h is called an *isomorphism* if there is a homomorphism $h' = (\tau', \rho') : S_2 \to S_1$ such that $hh' : S_2 \to S_2$ and $h'h : S_1 \to S_1$ are identities, i. e., $\tau\tau' : \Sigma_2^* \to \Sigma_2^*$, $\tau'\tau : \Sigma_1^* \to \Sigma_1^*$, $\rho\rho' : D_2 \to D_2$, and $\rho'\rho : D_1 \to D_1$ are identities. ■

Theorem (1.2.4.11): Let $S_1 = (\Sigma_1, D_1, x_1, \overline{}^1)$ and $S_1 = (\Sigma_2, D_2, x_2, \overline{}^2)$ be computation systems. Let $\tau : \Sigma_1^* \to \Sigma_2^*$ be a homomorphism and $\rho : D_1 \to D_2$ be a function. Then $h = (\tau, \rho)$ is an isomorphism from S_1 to S_2 if, and only if, τ and ρ are bijective and satisfy conditions (1.2.4.4.1) and (1.2.4.9.1).

Definition (1.2.4.12): A computation system $S = (\Sigma, D, x, \overline{})$ is said to be *reduced* if $D = R_S$. For each computation system $S = (\Sigma, D, x, \overline{})$, the computation system $\hat{S} = (\Sigma, R_S, x, \overline{})$ is called the *reduced form of* S, where for each $a \in \Sigma$, the partial function $\overline{a}$ in $\hat{S}$ is the restriction $\overline{a} \mid R_S$ of $\overline{a} \in S$. ■

Theorem (1.2.4.13): Let S_1 and S_2 be reduced computation systems. Let $h = (\tau, \rho) : S_1 \to S_2$ be a homomorphism. h is an isomorphism if, and only if, h is length preserving and bijective.

1.3 Equivalence of PFF–RDFD's and Turing Machines

In this section, we will show that PFF–RDFD's and Turing Machines are equivalent with respect to their computational power. Since we know that Turing Machines, Program Machines, and FIFO Petri

Nets are equivalent ([Min67], [FM82], and [MF85], respectively), it is sufficient to show that we can simulate each RDFD, and hence each PFF–RDFD, by a FIFO Petri Net and each Program Machine by a PFF–RDFD. Actually, the first simulation could be omitted and instead, we could simply invoke Church's Thesis to deduce that PFF–RDFD's have at most the computational power of Turing Machines (and thus of FIFO Petri Nets). However, we will show a little bit more than only the existence of a simulation, but also that the simulation of RDFD's (and thus of PFF–RDFD's) by FIFO Petri Nets is based on an isomorphism. For the other direction, a PFF–RDFD is sufficient to simulate a Program Machine.

Now, we will formally state our main result of this paper. The proof follows by the above and Theorems (1.3.1.1) and (1.3.2.1).

Theorem (1.3.1): PFF–RDFD's and Turing Machines have the same computational power.

1.3.1 Simulation of RDFD's by FIFO Petri Nets

Theorem (1.3.1.1): Every RDFD can be simulated by a FIFO Petri Net with respect to an isomorphism h.

Proof: Without loss of generality, we assume that our RDFD is given as a nf–RDFD (B_{RDFD}, $FLOWNAMES_{RDFD}$, $TYPES_{RDFD}$, P_{RDFD}, F_{RDFD}) with mappings *Enabled*, *Consume*, and *Produce* and with initial values $(bm_{initial}, r_{initial}, fs_{initial})$. We consider the computation system

$$S_{RDFD} = ((B_{RDFD}, \{C, P\}, I\!N), (bm, r, fs), (bm_{initial}, r_{initial}, fs_{initial}), \overset{RDFD}{\longrightarrow}).$$

We now construct a marked FIFO Petri Net $FPN = ((P_{FPN}, T_{FPN}, B_{FPN}, F_{FPN}, Q_{FPN}), M_{0,FPN})$ as follows:

For each flow in the RDFD we generate a place in the FIFO Petri Net. For an easier reference we assume that $\forall\, i = 1, \ldots, f, f_i \in FLOWNAMES_{RDFD} : FlowName(f_i) = f_i$. For each bubble in the RDFD we require additional places in the FIFO Petri Net to store the bubble's working mode and the values that have been read. Whenever we refer to type C_1, C_2, P_0, P_1, and P_2, we mean that the *Consume* or *Produce* case is in the related normal form, introduced in Definition (1.2.1.7).

$$\begin{aligned} P_{FPN} = \{&f_1, \ldots, f_f\} \\ &\cup \{b_{1,idle}, \ldots, b_{b,idle}\} \\ &\cup \bigcup_{i \in \{1,\ldots,b\}} \Big(\{b_{i,working:1} \mid Consume(b_i) \text{ is of type } C_1\} \end{aligned}$$

$$\cup\{b_{i,working:1}, \ldots, b_{i,working:m_i} \mid Consume(b_i) \text{ is of type } C_2,$$
$$m_i = (\#\text{ of cases in } Consume(b_i))\}\Big)$$

For each case (subcase) in the mappings $Consume$ and $Produce$, we generate a transition in the FIFO Petri Net.

$$T_{FPN} = \bigcup_{i\in\{1,\ldots,b\}} \Big(\{c_{i1} \mid Consume(b_i) \text{ is of type } C_1\}$$
$$\cup\{c_{i1}, \ldots, c_{im_i} \mid Consume(b_i) \text{ is of type } C_2,\ m_i = (\#\text{ of cases in } Consume(b_i))\}$$
$$\cup\{p_{i1}, \ldots, p_{is_i} \mid Produce(b_i) \text{ is of type } P_1,\ s_i = (\#\text{ of cases in } Produce(b_i))\}$$
$$\cup\{p_{i11}, \ldots, p_{i1l_{i1}}, \ldots, p_{im_i1}, \ldots, p_{im_il_{im_i}} \mid Produce(b_i) \text{ is of type } P_2,$$
$$m_i = (\#\text{ of cases in } Produce(b_i)),\ l_{ij} = (\#\text{ of subcases in case } j \text{ in } Produce(b_i))\}\Big)$$

For each $b_i \in B_{RDFD}$, define the mappings B_{FPN} and F_{FPN} in accordance with the mappings $Consume$ and $Produce$.

If $Consume(b_i)$ is of type C_1, then upon firing of transition c_{i1} the idle token I is removed from place $b_{i,idle}$ and the working token W is queued at place $b_{i,working:1}$. We define:

$$B(b_{i,idle}, c_{i1}) = I$$

$$F(c_{i1}, b_{i,working:1}) = W$$

If $Consume(b_i)$ is of type C_2, then upon firing of transition c_{ij} the idle token I is removed from place $b_{i,idle}$, the tokens $o^c_{ij1}, \ldots, o^c_{ijn_{ij1}}$ are removed from the places $f^c_{ij1}, \ldots, f^c_{ijn_{ij1}}$ representing consumable flows, the tokens $o^p_{ij1}, \ldots, o^p_{ijn_{ij2}}$ are removed from the places $f^p_{ij1}, \ldots, f^p_{ijn_{ij2}}$ representing persistent flows, the working token W is queued at place $b_{i,working:j}$ and the tokens $o^p_{ij1}, \ldots, o^p_{ijn_{ij2}}$ are queued at places $f^p_{ij1}, \ldots, f^p_{ijn_{ij2}}$ representing persistent flows. We define:

$$\begin{aligned} B(b_{i,idle}, c_{i1}) &= I \\ &\vdots \\ B(b_{i,idle}, c_{im_i}) &= I \end{aligned}$$

$$\begin{aligned} B(f^c_{i11}, c_{i1}) &= o^c_{i11} \\ &\vdots \\ B(f^c_{i1n_{i11}}, c_{i1}) &= o^c_{i1n_{i11}} \end{aligned}$$

$$
\begin{aligned}
B(f^p_{i11}, c_{i1}) &= o^p_{i11} \\
&\vdots \\
B(f^p_{i1n_{i12}}, c_{i1}) &= o^p_{i1n_{i12}} \\
&\vdots \\
B(f^c_{im_i1}, c_{im_i}) &= o^c_{im_i1} \\
&\vdots \\
B(f^c_{im_in_{im_i1}}, c_{im_i}) &= o^c_{im_in_{im_i1}} \\
B(f^p_{im_i1}, c_{im_i}) &= o^p_{im_i1} \\
&\vdots \\
B(f^p_{im_in_{im_i2}}, c_{im_i}) &= o^p_{im_in_{im_i2}}
\end{aligned}
$$

$$
\begin{aligned}
F(c_{i1}, b_{i,working:1}) &= W \\
&\vdots \\
F(c_{im_i}, b_{i,working:m_i}) &= W
\end{aligned}
$$

$$
\begin{aligned}
F(c_{i1}, f^p_{i11}) &= o^p_{i11} \\
&\vdots \\
F(c_{i1}, f^p_{i1n_{i12}}) &= o^p_{i1n_{i12}} \\
&\vdots \\
F(c_{im_i}, f^p_{im_i1}) &= o^p_{im_i1} \\
&\vdots \\
F(c_{im_i}, f^p_{im_in_{im_i2}}) &= o^p_{im_in_{im_i2}}
\end{aligned}
$$

If $Produce(b_i)$ is of type P_1, then upon firing of transition p_{ij} the working token W is removed from place $b_{i,working:j}$, the idle token I is queued at place $b_{i,idle}$, and the tokens $u_{ij1}, \ldots, u_{ijk_{ij}}$ are queued at places $h_{ij1}, \ldots, h_{ijk_{ij}}$. We define:

$$
\begin{aligned}
B(b_{i,working:1}, p_{i1}) &= W \\
&\vdots \\
B(b_{i,working:1}, p_{is_i}) &= W
\end{aligned}
$$

$$
\begin{aligned}
F(p_{i1}, b_{i,idle}) &= I \\
&\vdots \\
F(p_{is_i}, b_{i,idle}) &= I
\end{aligned}
$$

$$
\begin{aligned}
F(p_{i1}, h_{i11}) &= u_{i11} \\
&\vdots \\
F(p_{i1}, h_{i1k_{i1}}) &= u_{i1k_{i1}} \\
&\vdots \\
F(p_{is_i}, h_{is_i1}) &= u_{is_i1} \\
&\vdots \\
F(p_{is_i}, h_{is_ik_{is_i}}) &= u_{is_ik_{is_i}}
\end{aligned}
$$

If $Produce(b_i)$ is of type P_2, then upon firing of transition p_{ijk} the working token W is removed from place $b_{i,working:j}$, the idle token I is queued at place $b_{i,idle}$, and the tokens $u_{ijk1}, \ldots, u_{ijkq_{ijk}}$ are queued at places $h_{ijk1}, \ldots, h_{ijkq_{ijk}}$. Also, if place h_{ijk} represents a persistent flow, any value that is currently queued at this place will be removed upon firing of this transition. We define:

$$
\begin{aligned}
B(b_{i,working:1}, p_{i11}) &= W \\
&\vdots \\
B(b_{i,working:1}, p_{i1l_{i1}}) &= W \\
&\vdots \\
B(b_{i,working:m_i}, p_{im_i1}) &= W \\
&\vdots \\
B(b_{i,working:m_i}, p_{im_il_{im_i}}) &= W
\end{aligned}
$$

$$
\begin{aligned}
F(p_{i11}, b_{i,idle}) &= I \\
&\vdots \\
F(p_{i1l_{i1}}, b_{i,idle}) &= I \\
&\vdots \\
F(p_{im_i1}, b_{i,idle}) &= I
\end{aligned}
$$

$$
\begin{aligned}
&\vdots \\
F(p_{im_il_{im_i}}, b_{i,idle}) &= I \\
\\
F(p_{i11}, h_{i111}) &= u_{i111} \\
&\vdots \\
F(p_{i11}, h_{i11q_{i11}}) &= u_{i11q_{i11}} \\
&\vdots \\
F(p_{i1l_{i1}}, h_{i1l_{i1}1}) &= u_{i1l_{i1}1} \\
&\vdots \\
F(p_{i1l_{i1}}, h_{i1l_{i1}q_{i1l_{i1}}}) &= u_{i1l_{i1}q_{i1l_{i1}}} \\
&\vdots \\
F(p_{im_i1}, h_{im_i11}) &= u_{im_i11} \\
&\vdots \\
F(p_{im_i1}, h_{im_i1q_{im_i1}}) &= u_{im_i1q_{im_i1}} \\
&\vdots \\
F(p_{im_il_{im_i}}, h_{im_il_{im_i}1}) &= u_{im_il_{im_i}1} \\
&\vdots \\
F(p_{im_il_{im_i}}, h_{im_il_{im_i}q_{im_il_{im_i}}}) &= u_{im_il_{im_i}q_{im_il_{im_i}}} \\
\\
B(h_{i111}, p_{i11}) &= \text{any, if } \neg Consumable(h_{i111}) \\
&\vdots \\
B(h_{i11q_{i11}}, p_{i11}) &= \text{any, if } \neg Consumable(h_{i11q_{i11}}) \\
&\vdots \\
B(h_{im_il_{im_i}1}, p_{im_il_{im_i}}) &= \text{any, if } \neg Consumable(h_{im_il_{im_i}1}) \\
&\vdots \\
B(h_{im_il_{im_i}q_{im_il_{im_i}}}, p_{im_il_{im_i}}) &= \text{any, if } \neg Consumable(h_{im_il_{im_i}q_{im_il_{im_i}}})
\end{aligned}
$$

If nothing is produced for a particular combination of reads (case P_0), only the equations that involve $b_{i,working}$ and $b_{i,idle}$ are defined in that case.

The queue alphabet of the FIFO Petri Net is defined as follows:

$$Q_{FPN} = OBJECTS_{RDFD} \cup \{I, W\}$$

The initial marking $M_{0,FPN}$ is such that:

$$M_{0,FPN}(b_{i,idle}) = I \qquad \forall\, i \in \{1, \ldots, b\}$$

$$M_{0,FPN}(b_{i,working:j}) = <> \qquad \forall\, i \in \{1, \ldots, b\}\ \forall\, j \in \{1, \ldots, m_i\}$$

$$M_{0,FPN}(f_j) = fs(f_j) \qquad \forall\, j \in \{1, \ldots, f\}, f_j \in F_{RDFD}$$

Then, the computation system $S_{FPN} = (T_{FPN}, M_{FPN}, M_{0,FPN}, \overset{\frown}{}^{FPN})$ is realized by the FIFO Petri Net FPN. Consider the homomorphism $h = (\tau, \rho) : S_{RDFD} \rightarrow S_{FPN}$ where the homomorphism $\tau : (B_{RDFD}, \{C, P\}, I\!N)^* \rightarrow T^*_{FPN}$ and the injection $\rho : (bm, r, fs) \rightarrow M_{FPN}$ are defined as follows:

$$\tau((b_i, a, n)) = \begin{cases} c_{in}, & \text{if } a = C \\ p_{in}, & \text{if } a = P \text{ and } Produce(b_i) \text{ is of type } P_1 \\ p_{in_0 n_1}, & \text{if } a = P, Produce(b_i) \text{ is of type } P_2, \text{ and } n = \sum_{j=1}^{n_0 - 1} l_{ij} + n_1 \end{cases}$$

$\rho(bm, r, fs)$ is such that $\forall\, i \in \{1, \ldots, b\}\ \forall\, j \in \{1, \ldots, m_i\}$

$$M_{FPN}(b_{i,idle}) = \begin{cases} I, & \text{if } bm(b_i) = idle \\ <>, & \text{if } bm(b_i) = working \end{cases}$$

$$M_{FPN}(b_{i,working:j}) = \begin{cases} <>, & \text{if } bm(b_i) = idle \\ W, & \text{if } bm(b_i) = working, (Consume(b_i))_j \text{ has been executed} \\ <>, & \text{if } bm(b_i) = working, \exists\, k, k \neq j : (Consume(b_i))_k \text{ has been executed} \end{cases}$$

and $\forall\, j \in \{1, \ldots, f\}, f_j \in F_{RDFD}$

$$M_{FPN}(f_j) = fs(f_j).$$

Obviously, τ and ρ are bijective. Also, h satisfies (1.2.4.4.1) and (1.2.4.9.1). Hence, h is an isomorphism by Theorem (1.2.4.11). ■

According to the Definitions and Theorems from [KM82], here summarized in Subsection 1.2.4, the isomorphism h constructed in the previous proof is also a bijective (hence injective, surjective, hence spanning), length preserving, and principal homomorphism.

Example (1.3.1.2): We will provide a small example how to construct a FIFO Petri Net for a given nf–RDFD. Our nf–RDFD only consists of three bubbles A, B, and C, two consumable flows f and out, and two persistent flows g and h (see Figure 1.1). Bubble A only produces combinations of X

on flow f and 0 on flow g or Y on flow f and 1 on flow g. Bubble B can forward any value on flow h if it reads a 1 on flow g. It must forward the same value if it reads a 0 on flow g. Therefore, bubble C can read all possible combinations $X/0$, $X/1$, $Y/0$, and $Y/1$ on its flows f and h. Initially, flows g and h both contain a 0. All other flows are empty.

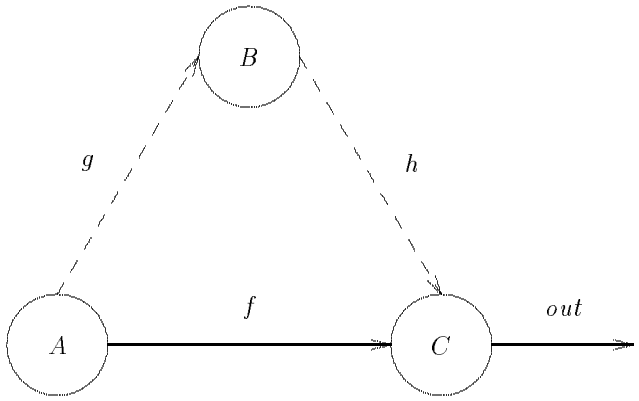

Figure 1.1: nf–RDFD.

The mappings *Enabled*, *Consume*, and *Produce* for the nf–RDFD shown in Figure 1.1 are specified as follows:

$$Enabled(A) = \lambda fs\ .\ true$$

$$Enabled(B) = \lambda fs\ .$$
$$\quad Head(fs(g)) = 0 \vee Head(fs(g)) = 1$$

$$Enabled(C) = \lambda fs\ .$$
$$\quad (\neg IsEmpty(f) \wedge Head(fs(f)) = X \wedge Head(fs(h)) = 0)$$
$$\quad \vee (\neg IsEmpty(f) \wedge Head(fs(f)) = X \wedge Head(fs(h)) = 1)$$
$$\quad \vee (\neg IsEmpty(f) \wedge Head(fs(f)) = Y \wedge Head(fs(h)) = 0)$$
$$\quad \vee (\neg IsEmpty(f) \wedge Head(fs(f)) = Y \wedge Head(fs(h)) = 1)$$

$$Consume(A) = \lambda(fs, r)\ .\ \{(fs, r)\}$$

$$Consume(B) = \lambda(fs, r)\ .$$
$$\quad \{\textbf{if}\ Head(fs(g)) = 0$$
$$\quad\ \textbf{then}\ In(g, B)(fs, r)$$
$$\quad\ \textbf{fi},$$
$$\quad\ \textbf{if}\ Head(fs(g)) = 1$$

$$
\begin{array}{l}
\quad \textbf{then } In(g,B)(fs,r) \\
\quad \textbf{fi} \\
\quad \} \\
Consume(C) = \lambda(fs,r)\ . \\
\quad \{\textbf{if } Head(fs(f)) = X \wedge Head(fs(h)) = 0 \\
\quad \textbf{then } In(f,C)(In(h,C)(fs,r)) \\
\quad \textbf{fi}, \\
\quad \textbf{if } Head(fs(f)) = X \wedge Head(fs(h)) = 1 \\
\quad \textbf{then } In(f,C)(In(h,C)(fs,r)) \\
\quad \textbf{fi}, \\
\quad \textbf{if } Head(fs(f)) = Y \wedge Head(fs(h)) = 0 \\
\quad \textbf{then } In(f,C)(In(h,C)(fs,r)) \\
\quad \textbf{fi}, \\
\quad \textbf{if } Head(fs(f)) = Y \wedge Head(fs(h)) = 1 \\
\quad \textbf{then } In(f,C)(In(h,C)(fs,r)) \\
\quad \textbf{fi} \\
\quad \} \\
\\
Produce(A) = \lambda(fs,r)\ . \\
\quad \{\, Out(X,f,A)(\,Out(0,g,A)(fs,r))\,, \\
\quad \ Out(Y,f,A)(Out(1,g,A)(fs,r))\} \\
\quad \} \\
Produce(B) = \lambda(fs,r)\ . \\
\quad \{\textbf{if } r(B)(g) = 0 \\
\quad \textbf{then } Out(0,h,B)(fs,r) \\
\quad \textbf{fi}, \\
\quad \textbf{if } r(B)(g) = 1 \\
\quad \textbf{then } Out(0,h,B)(fs,r) \\
\qquad \Box\, Out(1,h,B)(fs,r) \\
\quad \textbf{fi} \\
\quad \} \\
Produce(C) = \lambda(fs,r)\ .
\end{array}
$$

$\{$**if** $r(C)(f) = X \wedge r(C)(h) = 0$

then $Out(X0, out, C)(fs, r)$

fi,

if $r(C)(f) = X \wedge r(C)(h) = 1$

then $Out(X1, out, C)(fs, r)$

fi,

if $r(C)(f) = Y \wedge r(C)(h) = 0$

then $Out(Y0, out, C)(fs, r)$

fi,

if $r(C)(f) = Y \wedge r(C)(h) = 1$

then $Out(Y1, out, C)(fs, r)$

fi

$\}$

According to Theorem (1.3.1.1), the given nf–RDFD transforms into the following marked FIFO Petri Net $FPN = ((P_{FPN}, T_{FPN}, B_{FPN}, F_{FPN}, Q_{FPN}), M_{0,FPN})$:

$$P_{FPN} = \{f, g, h, out\} \cup \{A_i, A_{w:1}, B_i, B_{w:1}, B_{w:2}, C_i, C_{w:1}, C_{w:2}, C_{w:3}, C_{w:4}\}$$

$$T_{FPN} = \{C_{A1}, C_{B1}, C_{B2}, C_{C1}, C_{C2}, C_{C3}, C_{C4}\}$$
$$\cup\{P_{A1}, P_{A2}, P_{B11}, P_{B21}, P_{B22}, P_{C11}, P_{C21}, P_{C31}, P_{C41}\}$$

The initial marking $M_{0,FPN}$ is such that:

$$M_{0,FPN}(A_i) = M_{0,FPN}(B_i) = M_{0,FPN}(C_i) = I$$

$$M_{0,FPN}(A_{w:1}) = M_{0,FPN}(B_{w:1}) = M_{0,FPN}(B_{w:2}) = <>$$

$$M_{0,FPN}(C_{w:1}) = M_{0,FPN}(C_{w:2}) = M_{0,FPN}(B_{w:3}) = M_{0,FPN}(B_{w:4}) = <>$$

$$M_{0,FPN}(f) = <>$$

$$M_{0,FPN}(g) = 0$$

$$M_{0,FPN}(h) = 0$$

$$M_{0,FPN}(out) = <>$$

B_{FPN}, F_{FPN}, and Q_{FPN} can be gained from Figure 1.2. In this figure, we use a double arrow to indicate that the forward and backward incidence mappings for a particular place/transition combination are identical, i. e., $B(g, C_{B1}) = F(C_{B1}, g) = 0$. This means, that transition C_{B1} (if fired) removes the head element (i. e., 0) of place g and appends the same element to this place. Obviously, place g always contains only one element since it represents a persistent flow of the nf–RDFD. ■

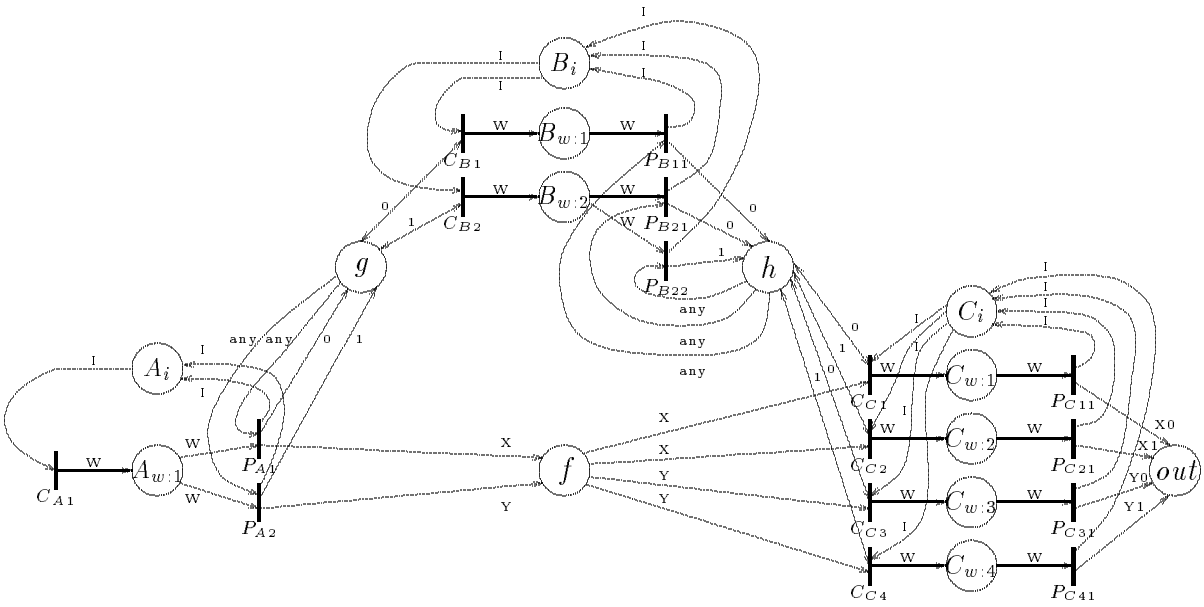

Figure 1.2: FIFO Petri Net.

1.3.2 Simulation of Program Machines by PFF–RDFD's

Theorem (1.3.2.1): Every Program Machine can be simulated by an PFF–RDFD.

Proof: Assume we have a Program Machine with registers $R = \{r_1, \ldots, r_p\}$, labels $Q = \{q_0, \ldots, q_r\}$ and instructions I. We will partially follow the proof of how to simulate a Program Machine by a FIFO Petri Net (given in [FM82] and [MF85]) when we indicate how to simulate a Program Machine by a PFF–RDFD.

We define our PFF–RDFD $RDFD_{PM}$, given in the normal form of Definition (1.2.1.7), as follows:

$$RDFD_{PM} = (B_{RDFD}, FLOWNAMES_{RDFD}, TYPES_{RDFD}, P_{RDFD}, F_{RDFD}),$$

where

$B_{RDFD} = \{q_0, \ldots, q_r\} \cup \{r_1, \ldots, r_p\}$

$FLOWNAMES_{RDFD} = \{do\_q_0, do\_q_0\_q_1\}$

$$\cup \bigcup_{s \in I} \{do\_q_s\_q_m \mid \text{instruction labeled } q_s \text{ is an increment instruction}\}$$

$$\cup \bigcup_{s \in I} \{do\_q_s\_q_m, do\_q_s\_q_l \mid \text{instruction labeled } q_s \text{ is a test and decrement instruction}\}$$

$$\cup \bigcup_{s \in I} \{i\_q_s\_r_i, o\_r_i\_q_s \mid \text{instruction labeled } q_s \text{ accesses register } r_i\}$$

$$\cup \bigcup_{i=1,\ldots,r} \{next_i, act_i, val_i\}$$

$TYPES_{RDFD} = \text{ENAB} \cup \text{ACTION} \cup \text{RESULT} \cup \text{FROM} \cup \text{YES} \cup \text{BIT}$

where $\text{ENAB} = \{start, go\}$, $\text{ACTION} = \{add, sub\}$, $\text{RESULT} = \{done, iszero\}$,

$\text{FROM} = \{1, \ldots, r\}$, $\text{YES} = \{yes\}$, and $\text{BIT} = \{0, 1\}$

$P_{RDFD} = \{consumable\}$

$F_{RDFD} = \{(q_0, do\_q_0, \text{ENAB}, q_0, consumable), (q_0, do\_q_0\_q_1, \text{ENAB}, q_1, consumable)\}$

$\cup\{(q_s, do\_q_s\_q_m, \text{ENAB}, q_m, consumable) \mid \forall\, FLOWNAMES\ do\_q_s\_q_m\}$

$\cup\{(q_s, i\_q_s\_r_i, \text{ACTION}, r_i, consumable) \mid \forall\, FLOWNAMES\ i\_q_s\_r_i\}$

$\cup\{(r_i, o\_r_i\_q_s, \text{RESULT}, q_s, consumable) \mid \forall\, FLOWNAMES\ o\_r_i\_q_s\}$

$$\cup \bigcup_{i=1,\ldots,r} \{(r_i, next_i, \text{FROM}, r_i, consumable),$$
$$(r_i, act_i, \text{YES}, r_i, consumable),$$
$$(r_i, val_i, \text{BIT}, r_i, consumable)\}$$

We use a unary representation for nonnegative integers which are the only possible contents of the registers of the simulated Program Machine. The value 0 stands for 0 and the sequence of values $(\underbrace{1, \ldots, 1}_{n}, 0)$ represents n. Initially, the values on the flows are *start* on do_q_0 and the unary representation of the initial contents of the registers $r_1, \ldots, r_p$, i. e., the values $k_1^1, \ldots, k_1^p$, on the flows $val_1, \ldots, val_p$, respectively. All other flows are empty.

The mapping *Enabled* is defined as follows:

If the bubble represents the start instruction (see Figure 1.3):

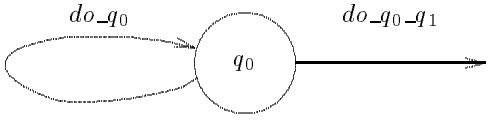

Figure 1.3: Bubble q_0.

$$Enabled(q_0) = \lambda fs\ .\ (\neg IsEmpty(do\_q_0) \wedge Head(fs(do\_q_0)) = start)$$

If the instruction labeled q_s is an increment instruction accessing register r_i and its corresponding bubble has inflows $do\_q_{m_1}\_q_s, \ldots, do\_q_{m_s}\_q_s, o\_r_i\_q_s$ (see Figure 1.4):

$$Enabled(q_s) = \lambda fs\ .$$

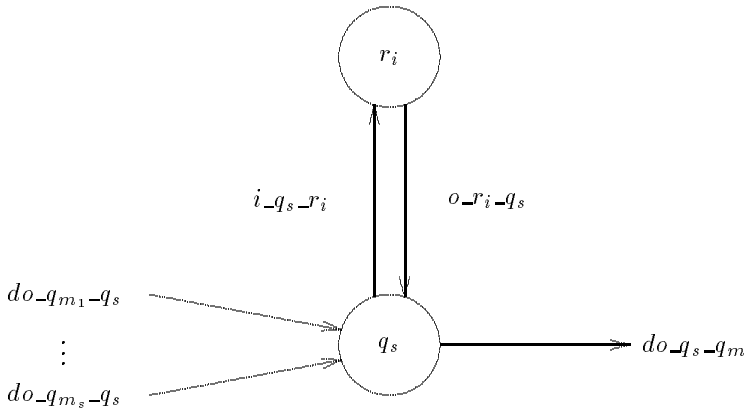

Figure 1.4: Bubble q_s for Increment Instruction.

$$(\neg IsEmpty(do\_q_{m_1}\_q_s) \wedge Head(fs(do\_q_{m_1}\_q_s)) = go)$$
$$\vee \ldots \vee$$
$$(\neg IsEmpty(do\_q_{m_s}\_q_s) \wedge Head(fs(do\_q_{m_s}\_q_s)) = go)$$
$$\vee (\neg IsEmpty(o\_r_i\_q_s) \wedge Head(fs(o\_r_i\_q_s)) = done)$$

If the instruction labeled q_s is a test and decrement instruction accessing register r_i and its corresponding bubble has inflows $do\_q_{m_1}\_q_s, \ldots, do\_q_{m_s}\_q_s, o\_r_i\_q_s$ (see Figure 1.5):

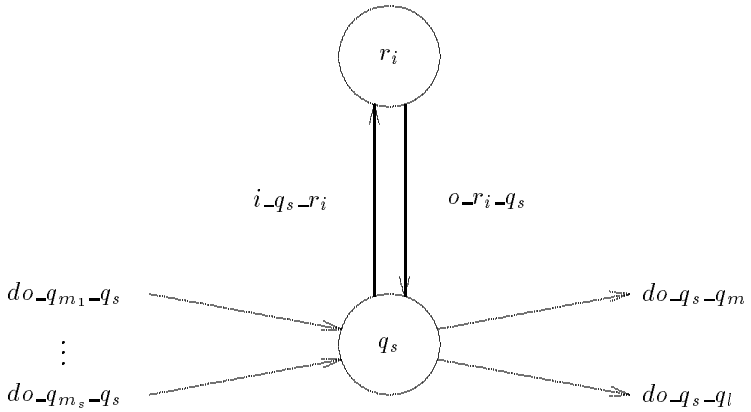

Figure 1.5: Bubble q_s for Test and Decrement Instruction.

$$Enabled(q_s) = \lambda fs\ .$$
$$(\neg IsEmpty(do\_q_{m_1}\_q_s) \wedge Head(fs(do\_q_{m_1}\_q_s)) = go)$$
$$\vee \ldots \vee$$
$$(\neg IsEmpty(do\_q_{m_s}\_q_s) \wedge Head(fs(do\_q_{m_s}\_q_s)) = go)$$

$\vee(\neg IsEmpty(o\_r_i\_q_s) \wedge Head(fs(o\_r_i\_q_s)) = done)$

$\vee(\neg IsEmpty(o\_r_i\_q_s) \wedge Head(fs(o\_r_i\_q_s)) = iszero)$

If the instruction labeled q_s is the halt instruction, i. e., $q_s = q_f$, and its corresponding bubble has inflows $do\_q_{m_1}\_q_f, \ldots, do\_q_{m_f}\_q_f$ (see Figure 1.6):

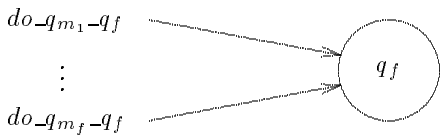

Figure 1.6: Bubble q_f.

$Enabled(q_f) = \lambda fs\ .\ false$

If the bubble representing register r_i has inflows $i\_q_{s_1}\_r_i, \ldots, i\_q_{s_i}\_r_i, next_i, act_i, val_i$ (see Figure 1.7):

$Enabled(r_i) = \lambda fs\ .$

[These lines are required for add:]

$(\neg IsEmpty(i\_q_{s_1}\_r_i) \wedge Head(fs(i\_q_{s_1}\_r_i)) = add)$

$\vee \ldots \vee$

$(\neg IsEmpty(i\_q_{s_i}\_r_i) \wedge Head(fs(i\_q_{s_i}\_r_i)) = add)$

[These lines are required for sub:]

$\vee(\neg IsEmpty(i\_q_{s_1}\_r_i) \wedge Head(fs(i\_q_{s_1}\_r_i)) = sub$

$\quad \wedge \neg IsEmpty(val_i) \wedge Head(fs(val_i)) = 0)$

$\vee \ldots \vee$

$(\neg IsEmpty(i\_q_{s_i}\_r_i) \wedge Head(fs(i\_q_{s_i}\_r_i)) = sub$

$\quad \wedge \neg IsEmpty(val_i) \wedge Head(fs(val_i)) = 0)$

$\vee(\neg IsEmpty(i\_q_{s_1}\_r_i) \wedge Head(fs(i\_q_{s_1}\_r_i)) = sub$

$\quad \wedge \neg IsEmpty(val_i) \wedge Head(fs(val_i)) = 1)$

$\vee \ldots \vee$

$(\neg IsEmpty(i\_q_{s_i}\_r_i) \wedge Head(fs(i\_q_{s_i}\_r_i)) = sub$

$\quad \wedge \neg IsEmpty(val_i) \wedge Head(fs(val_i)) = 1)$

[This line is required to copy the tail of 1's:]

$\vee(\neg IsEmpty(act_i) \wedge Head(fs(act_i)) = yes$

$\quad \wedge \neg IsEmpty(val_i) \wedge Head(fs(val_i)) = 1)$

[These lines are required to activate the next instruction:]

$$\vee(\neg IsEmpty(act_i) \wedge Head(fs(act_i)) = yes$$
$$\wedge\neg IsEmpty(val_i) \wedge Head(fs(val_i)) = 0$$
$$\wedge\neg IsEmpty(next_i) \wedge Head(fs(next_i)) = s_1)$$
$$\vee \ldots \vee$$
$$(\neg IsEmpty(act_i) \wedge Head(fs(act_i)) = yes$$
$$\wedge\neg IsEmpty(val_i) \wedge Head(fs(val_i)) = 0$$
$$\wedge\neg IsEmpty(next_i) \wedge Head(fs(next_i)) = s_i)$$

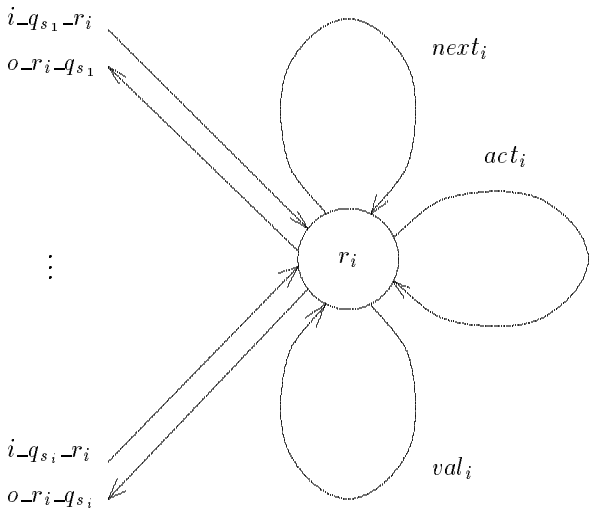

Figure 1.7: Bubble r_i.

As usual, we assume that whenever a set of input conditions on flows enables a bubble, all the head elements on these flows will be consumed when this bubble actually goes from *idle* to *working*. Therefore, we omit to list the mapping *Consume*. We immediately continue with *Produce*.

If the bubble represents the start instruction (see Figure 1.3):

$$Produce(q_0) = \lambda(fs, r)\ .$$
$$\{\textbf{if}\ r(q_0)(do\_q_0) = start$$
$$\textbf{then}\ Out(go, do\_q_0\_q_1, q_0)(fs, r)$$
$$\textbf{fi}$$
$$\}$$

If the instruction labeled q_s is an increment instruction accessing register r_i and its corresponding bubble has inflows $do\_q_{m_1}\_q_s, \ldots, do\_q_{m_s}\_q_s, o\_r_i\_q_s$ and outflows $do\_q_s\_q_m, i\_q_s\_r_i$ (see Figure 1.4):

$Produce(q_s) = \lambda(fs, r)$.

{**if** $r(q_s)(do\_q_{m_1}\_q_s) = go$

then $Out(add, i\_q_s\_r_i, q_s)(fs, r)$

fi,

...,

if $r(q_s)(do\_q_{m_s}\_q_s) = go$

then $Out(add, i\_q_s\_r_i, q_s)(fs, r)$

fi,

if $r(q_s)(o\_r_i\_q_s) = done$

then $Out(go, do\_q_s\_q_m, q_s)(fs, r)$

fi

}

If the instruction labeled q_s is a test and decrement instruction accessing register r_i and its corresponding bubble has inflows $do\_q_{m_1}\_q_s, \ldots, do\_q_{m_s}\_q_s, o\_r_i\_q_s$ and outflows $do\_q_s\_q_m, do\_q_s\_q_l, i\_q_s\_r_i$ (see Figure 1.5):

$Produce(q_s) = \lambda(fs, r)$.

{**if** $r(q_s)(do\_q_{m_1}\_q_s) = go$

then $Out(sub, i\_q_s\_r_i, q_s)(fs, r)$

fi,

...,

if $r(q_s)(do\_q_{m_s}\_q_s) = go$

then $Out(sub, i\_q_s\_r_i, q_s)(fs, r)$

fi,

if $r(q_s)(o\_r_i\_q_s) = done$

then $Out(go, do\_q_s\_q_m, q_s)(fs, r)$

fi,

if $r(q_s)(o\_r_i\_q_s) = iszero$

then $Out(go, do\_q_s\_q_l, q_s)(fs, r)$

fi

}

The next line is given only for formal reasons, but it will never be executed since the corresponding enabling condition is always *false*. If the instruction labeled q_s is the halt instruction, i. e., $q_s = q_f$,

and its corresponding bubble has inflows $do\_q_{m_1}\_q_f, \ldots, do\_q_{m_f}\_q_f$ (see Figure 1.6):

$$Produce(q_f) = \lambda(fs, r)\ .\ \{(fs, [q_f \mapsto \lambda f\ .\ \bot]r)\}$$

If the bubble representing register r_i has inflows $i\_q_{s_1}\_r_i, \ldots, i\_q_{s_i}\_r_i, next_i, act_i, val_i$ and outflows $o\_r_i\_q_{s_1}, \ldots, o\_r_i\_q_{s_i}, next_i, act_i, val_i$ (see Figure 1.7):

$$Produce(r_i) = \lambda(fs, r)\ .$$

[These lines are required for add:]

$$\{\textbf{if}\ r(r_i)(i\_q_{s_1}\_r_i) = add$$

$$\textbf{then}\ Out(s_1, next_i, r_i)(Out(yes, act_i, r_i)(Out(1, val_i, r_i)(fs, r)))$$

$$\textbf{fi},$$

$$\ldots,$$

$$\textbf{if}\ r(r_i)(i\_q_{s_i}\_r_i) = add$$

$$\textbf{then}\ Out(s_i, next_i, r_i)(Out(yes, act_i, r_i)(Out(1, val_i, r_i)(fs, r)))$$

$$\textbf{fi},$$

[These lines are required for sub:]

$$\textbf{if}\ r(r_i)(i\_q_{s_1}\_r_i) = sub \wedge r(r_i)(val_i) = 0$$

$$\textbf{then}\ Out(0, val_i, r_i)(Out(iszero, o\_r_i\_q_{s_1}, r_i)(fs, r))$$

$$\textbf{fi},$$

$$\ldots,$$

$$\textbf{if}\ r(r_i)(i\_q_{s_i}\_r_i) = sub \wedge r(r_i)(val_i) = 0$$

$$\textbf{then}\ Out(0, val_i, r_i)(Out(iszero, o\_r_i\_q_{s_i}, r_i)(fs, r))$$

$$\textbf{fi},$$

$$\textbf{if}\ r(r_i)(i\_q_{s_1}\_r_i) = sub \wedge r(r_i)(val_i) = 1$$

$$\textbf{then}\ Out(s_1, next_i, r_i)(Out(yes, act_i, r_i)(fs, r))$$

$$\textbf{fi},$$

$$\ldots,$$

$$\textbf{if}\ r(r_i)(i\_q_{s_i}\_r_i) = sub \wedge r(r_i)(val_i) = 1$$

$$\textbf{then}\ Out(s_i, next_i, r_i)(Out(yes, act_i, r_i)(fs, r))$$

$$\textbf{fi},$$

[This line is required to copy the tail of 1's:]

$$\textbf{if}\ r(r_i)(act_i) = yes \wedge r(r_i)(val_i) = 1$$

$$\textbf{then}\ Out(yes, act_i, r_i)(Out(1, val_i, r_i)(fs, r))$$

$$\textbf{fi},$$

[These lines are required to activate the next instruction:]

if $r(r_i)(act_i) = yes \wedge r(r_i)(val_i) = 0 \wedge r(r_i)(next_i) = s_1$

then $Out(0, val_i, r_i)(Out(done, o\_r_i\_q_{s_1}, r_i)(fs, r))$

fi,

$\ldots$,

if $r(r_i)(act_i) = yes \wedge r(r_i)(val_i) = 0 \wedge r(r_i)(next_i) = s_i$

then $Out(0, val_i, r_i)(done, Out(o\_r_i\_q_{s_i}, r_i)(fs, r))$

fi

}

■

Example (1.3.2.2): Consider a Program Machine that initially has some nonnegative values in its registers r_1 and r_2, while r_3 contains 0. The following code stores the sum of r_1 and r_2 in r_3 and erases the original values:

q_0 : **start goto** q_1;

q_1 : **if** $r_1 = 0$ **then goto** q_3 **else** $r_1 := r_1 - 1$ **goto** q_2;

q_2 : $r_3 := r_3 + 1$ **goto** q_1;

q_3 : **if** $r_2 = 0$ **then goto** q_f **else** $r_2 := r_2 - 1$ **goto** q_4;

q_4 : $r_3 := r_3 + 1$ **goto** q_3;

q_f : **halt.**

The graphical representation of the equivalent PFF–RDFD is given in Figure 1.8.

Based on this diagram, the formal definitions can be gained according to Theorem (1.3.2.1) as:

$B_{RDFD} = \{q_0, q_1, q_2, q_3, q_4, q_f\} \cup \{r_1, r_2, r_3\}$

$FLOWNAMES_{RDFD} = \{do\_q_0, do\_q_0\_q_1\}$

$\cup\{do\_q_2\_q_1, do\_q_4\_q_3\}$

$\cup\{do\_q_1\_q_3, do\_q_1\_q_2, do\_q_3\_q_f, do\_q_3\_q_4\}$

$\cup\{i\_q_1\_r_1, o\_r_1\_q_1, i\_q_2\_r_3, o\_r_3\_q_2, i\_q_3\_r_2, o\_r_2\_q_3, i\_q_4\_r_3, o\_r_3\_q_4\}$

$\cup\{next_1, act_1, val_1, next_2, act_2, val_2, next_3, act_3, val_3\}$

$TYPES_{RDFD}$ (with FROM $= \{1, 2, 3, 4\}$), P_{RDFD}, and F_{RDFD} follow immediately.

Assume we want to add 2 and 3, then the initial values, i. e., sequences of values, on the flows are $do\_q_0 = start$, $val_1 = (1, 1, 0)$, $val_2 = (1, 1, 1, 0)$ and $val_3 = 0$. All other flows are empty.

The mapping *Enabled* is defined as follows:

$Enabled(q_0) = \lambda fs$.

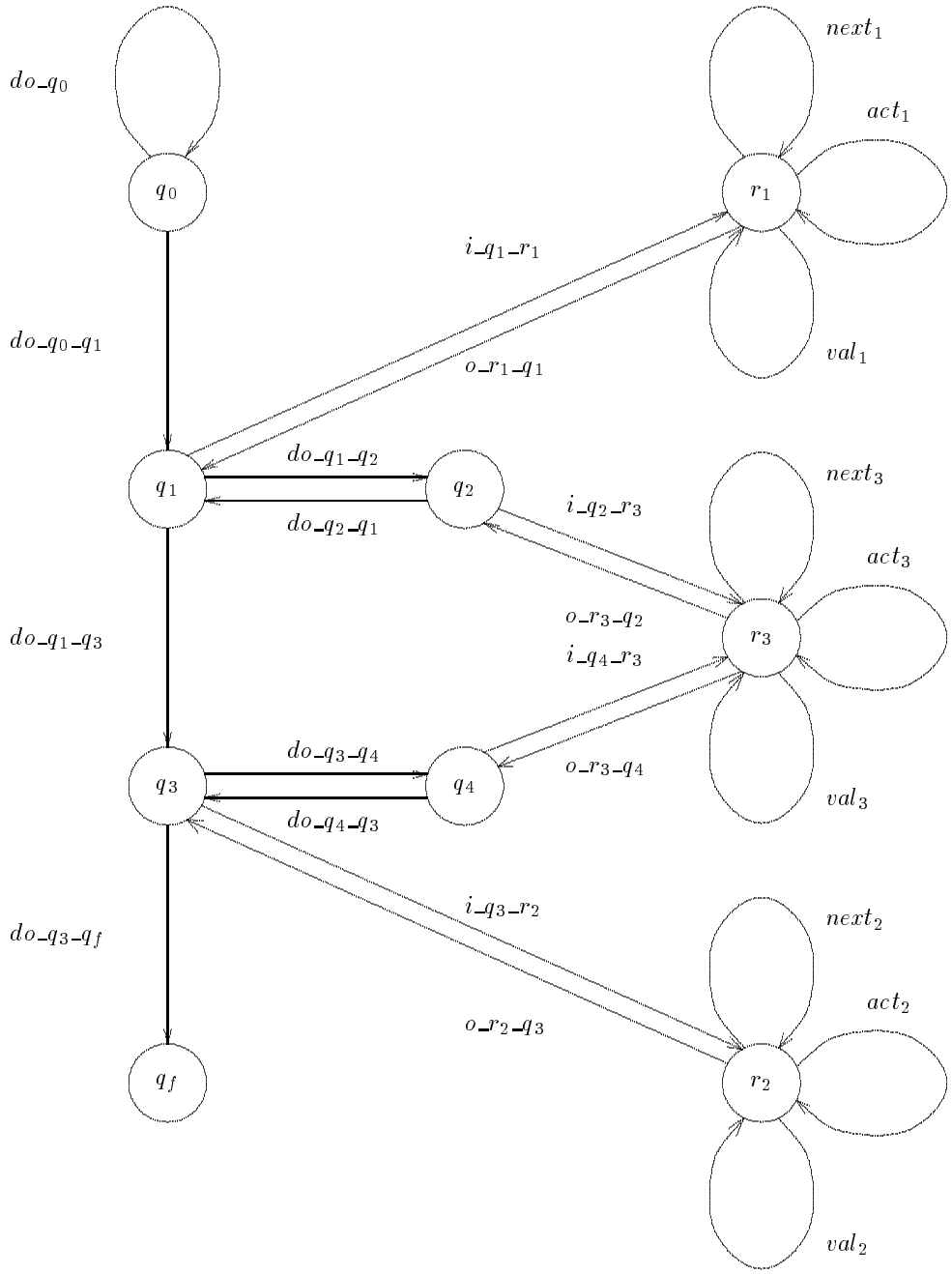

Figure 1.8: PFF–RDFD.

$$(\neg IsEmpty(do\_q_0) \wedge Head(fs(do\_q_0)) = start)$$

$$Enabled(q_1) = \lambda fs\ .$$

$$(\neg IsEmpty(do\_q_0\_q_1) \wedge Head(fs(do\_q_0\_q_1)) = go)$$

$$\vee(\neg IsEmpty(do\_q_2\_q_1) \wedge Head(fs(do\_q_2\_q_1)) = go)$$

$$\vee(\neg IsEmpty(o\_r_1\_q_1) \wedge Head(fs(o\_r_1\_q_1)) = done)$$

$$\vee(\neg IsEmpty(o\_r_1\_q_1) \wedge Head(fs(o\_r_1\_q_1)) = iszero)$$

$Enabled(q_2) = \lambda fs\ .$

$$(\neg IsEmpty(do\_q_1\_q_2) \wedge Head(fs(do\_q_1\_q_2)) = go)$$

$$\vee(\neg IsEmpty(o\_r_3\_q_2) \wedge Head(fs(o\_r_3\_q_2)) = done)$$

$Enabled(q_3) = \lambda fs\ .$

$$(\neg IsEmpty(do\_q_1\_q_3) \wedge Head(fs(do\_q_1\_q_3)) = go)$$

$$\vee(\neg IsEmpty(do\_q_4\_q_3) \wedge Head(fs(do\_q_4\_q_3)) = go)$$

$$\vee(\neg IsEmpty(o\_r_2\_q_3) \wedge Head(fs(o\_r_2\_q_3)) = done)$$

$$\vee(\neg IsEmpty(o\_r_2\_q_3) \wedge Head(fs(o\_r_2\_q_3)) = iszero)$$

$Enabled(q_4) = \lambda fs\ .$

$$(\neg IsEmpty(do\_q_3\_q_4) \wedge Head(fs(do\_q_3\_q_4)) = go)$$

$$\vee(\neg IsEmpty(o\_r_3\_q_4) \wedge Head(fs(o\_r_3\_q_4)) = done)$$

$Enabled(q_f) = \lambda fs\ .\ false$

$Enabled(r_1) = \lambda fs\ .$

$$(\neg IsEmpty(i\_q_1\_r_1) \wedge Head(fs(i\_q_1\_r_1)) = add)$$

$$\vee(\neg IsEmpty(i\_q_1\_r_1) \wedge Head(fs(i\_q_1\_r_1)) = sub$$

$$\wedge\neg IsEmpty(val_1) \wedge Head(fs(val_1)) = 0)$$

$$\vee(\neg IsEmpty(i\_q_1\_r_1) \wedge Head(fs(i\_q_1\_r_1)) = sub$$

$$\wedge\neg IsEmpty(val_1) \wedge Head(fs(val_1)) = 1)$$

$$\vee(\neg IsEmpty(act_1) \wedge Head(fs(act_1)) = yes$$

$$\wedge\neg IsEmpty(val_1) \wedge Head(fs(val_1)) = 1)$$

$$\vee(\neg IsEmpty(act_1) \wedge Head(fs(act_1)) = yes$$

$$\wedge\neg IsEmpty(val_1) \wedge Head(fs(val_1)) = 0$$

$$\wedge\neg IsEmpty(next_1) \wedge Head(fs(next_1)) = 1)$$

$Enabled(r_2) = \lambda fs\ .$

$$(\neg IsEmpty(i\_q_3\_r_2) \wedge Head(fs(i\_q_3\_r_2)) = add)$$

$$\vee(\neg IsEmpty(i\_q_3\_r_2) \wedge Head(fs(i\_q_3\_r_2)) = sub$$

$$\wedge\neg IsEmpty(val_2) \wedge Head(fs(val_2)) = 0)$$

$$\vee(\neg IsEmpty(i\_q_3\_r_2) \wedge Head(fs(i\_q_3\_r_2)) = sub$$

$$\wedge\neg IsEmpty(val_2) \wedge Head(fs(val_2)) = 1)$$

$$\vee(\neg IsEmpty(act_2) \wedge Head(fs(act_2)) = yes$$

$$\wedge \neg IsEmpty(val_2) \wedge Head(fs(val_2)) = 1)$$
$$\vee (\neg IsEmpty(act_2) \wedge Head(fs(act_2)) = yes$$
$$\wedge \neg IsEmpty(val_2) \wedge Head(fs(val_2)) = 0$$
$$\wedge \neg IsEmpty(next_2) \wedge Head(fs(next_2)) = 3)$$

$Enabled(r_3) = \lambda fs$.

$$(\neg IsEmpty(i\_q_2\_r_3) \wedge Head(fs(i\_q_2\_r_3)) = add)$$
$$\vee (\neg IsEmpty(i\_q_4\_r_3) \wedge Head(fs(i\_q_4\_r_3)) = add)$$
$$\vee (\neg IsEmpty(i\_q_2\_r_3) \wedge Head(fs(i\_q_2\_r_3)) = sub$$
$$\wedge \neg IsEmpty(val_3) \wedge Head(fs(val_3)) = 0)$$
$$\vee (\neg IsEmpty(i\_q_4\_r_3) \wedge Head(fs(i\_q_4\_r_3)) = sub$$
$$\wedge \neg IsEmpty(val_3) \wedge Head(fs(val_3)) = 0)$$
$$\vee (\neg IsEmpty(i\_q_2\_r_3) \wedge Head(fs(i\_q_2\_r_3)) = sub$$
$$\wedge \neg IsEmpty(val_3) \wedge Head(fs(val_3)) = 1)$$
$$\vee (\neg IsEmpty(i\_q_4\_r_3) \wedge Head(fs(i\_q_4\_r_3)) = sub$$
$$\wedge \neg IsEmpty(val_3) \wedge Head(fs(val_3)) = 1)$$
$$\vee (\neg IsEmpty(act_3) \wedge Head(fs(act_3)) = yes$$
$$\wedge \neg IsEmpty(val_3) \wedge Head(fs(val_3)) = 1)$$
$$\vee (\neg IsEmpty(act_3) \wedge Head(fs(act_3)) = yes$$
$$\wedge \neg IsEmpty(val_3) \wedge Head(fs(val_3)) = 0$$
$$\wedge \neg IsEmpty(next_3) \wedge Head(fs(next_3)) = 2)$$
$$\vee (\neg IsEmpty(act_3) \wedge Head(fs(act_3)) = yes$$
$$\wedge \neg IsEmpty(val_3) \wedge Head(fs(val_3)) = 0$$
$$\wedge \neg IsEmpty(next_3) \wedge Head(fs(next_3)) = 4)$$

As usual, we assume that whenever a set of input conditions on flows enables a bubble, all the head elements on these flows will be consumed when this bubble actually goes from *idle* to *working*. Therefore, we omit to list the mapping *Consume*. We immediately continue with the mapping *Produce*.

$Produce(q_0) = \lambda(fs, r)$.

$\{$**if** $r(q_0)(do\_q_0) = start$

then $Out(go, do\_q_0\_q_1, q_0)(fs, r)$

fi

$\}$

$Produce(q_1) = \lambda(fs, r)$.

$\{$**if** $r(q_1)(do\_q_0\_q_1) = go$

then $Out(sub, i\_q_1\_r_1, q_1)(fs, r)$

fi,

if $r(q_1)(do\_q_2\_q_1) = go$

then $Out(sub, i\_q_1\_r_1, q_1)(fs, r)$

fi,

if $r(q_1)(o\_r_1\_q_1) = done$

then $Out(go, do\_q_1\_q_2, q_1)(fs, r)$

fi,

if $r(q_1)(o\_r_1\_q_1) = iszero$

then $Out(go, do\_q_1\_q_3, q_1)(fs, r)$

fi

$\}$

$Produce(q_2) = \lambda(fs, r)$.

$\{$**if** $r(q_2)(do\_q_1\_q_2)) = go$

then $Out(add, i\_q_2\_r_3, q_2)(fs, r)$

fi,

if $r(q_2)(o\_r_3\_q_2)) = done$

then $Out(go, do\_q_2\_q_1, q_2)(fs, r)$

fi

$\}$

$Produce(q_3) = \lambda(fs, r)$.

$\{$**if** $r(q_3)(do\_q_1\_q_3) = go$

then $Out(sub, i\_q_3\_r_2, q_3)(fs, r)$

fi,

if $r(q_3)(do\_q_4\_q_3) = go$

then $Out(sub, i\_q_3\_r_2, q_3)(fs, r)$

fi,

if $r(q_3)(o\_r_2\_q_3) = done$

then $Out(go, do\_q_3\_q_4, q_3)(fs, r)$

fi,

if $r(q_3)(o\_r_2\_q_3) = iszero$

$\quad$ **then** $Out(go, do\_q_3\_q_f, q_3)(fs, r)$

$\quad$ **fi**

$\quad\}$

$Produce(q_4) = \lambda(fs, r)$.

$\quad\{$**if** $r(q_4)(do\_q_3\_q_4)) = go$

$\quad$ **then** $Out(add, i\_q_4\_r_3, q_4)(fs, r)$

$\quad$ **fi**,

$\quad$ **if** $r(q_4)(o\_r_3\_q_4)) = done$

$\quad$ **then** $Out(go, do\_q_4\_q_3, q_4)(fs, r)$

$\quad$ **fi**

$\quad\}$

$Produce(q_f) = \lambda(fs, r)$. $\{(fs, [q_f \mapsto \lambda f$. $\bot]r)\}$

$Produce(r_1) = \lambda(fs, r)$.

$\quad\{$**if** $r(r_1)(i\_q_1\_r_1) = add$

$\quad$ **then** $Out(1, next_1, r_1)(Out(yes, act_1, r_1)(Out(1, val_1, r_1)(fs, r)))$

$\quad$ **fi**,

$\quad$ **if** $r(r_1)(i\_q_1\_r_1) = sub \wedge r(r_1)(val_1) = 0$

$\quad$ **then** $Out(0, val_1, r_1)(Out(iszero, o\_r_1\_q_1, r_1)(fs, r))$

$\quad$ **fi**,

$\quad$ **if** $r(r_1)(i\_q_1\_r_1) = sub \wedge r(r_1)(val_1) = 1$

$\quad$ **then** $Out(1, next_1, r_1)(Out(yes, act_1, r_1)(fs, r))$

$\quad$ **fi**,

$\quad$ **if** $r(r_1)(act_1) = yes \wedge r(r_1)(val_1) = 1$

$\quad$ **then** $Out(yes, act_1, r_1)(Out(1, val_1, r_1)(fs, r))$

$\quad$ **fi**,

$\quad$ **if** $r(r_1)(act_1) = yes \wedge r(r_1)(val_1) = 0 \wedge r(r_1)(next_1) = 1$

$\quad$ **then** $Out(0, val_1, r_1)(Out(done, o\_r_1\_q_1, r_1)(fs, r))$

$\quad$ **fi**

$\quad\}$

$Produce(r_2) = \lambda(fs, r)$.

$\quad\{$**if** $r(r_2)(i\_q_3\_r_2) = add$

$\quad$ **then** $Out(3, next_2, r_2)(Out(yes, act_2, r_2)(Out(1, val_2, r_2)(fs, r)))$

$$
\begin{array}{l}
\quad \textbf{fi}, \\
\quad \textbf{if } r(r_2)(i\_q_3\_r_2) = sub \wedge r(r_2)(val_2) = 0 \\
\quad \textbf{then } Out(0, val_2, r_2)(Out(iszero, o\_r_2\_q_3, r_2)(fs, r)) \\
\quad \textbf{fi}, \\
\quad \textbf{if } r(r_2)(i\_q_3\_r_2) = sub \wedge r(r_2)(val_2) = 1 \\
\quad \textbf{then } Out(3, next_2, r_2)(Out(yes, act_2, r_2)(fs, r)) \\
\quad \textbf{fi}, \\
\quad \textbf{if } r(r_2)(act_2) = yes \wedge r(r_2)(val_2) = 1 \\
\quad \textbf{then } Out(yes, act_2, r_2)(Out(1, val_2, r_2)(fs, r)) \\
\quad \textbf{fi}, \\
\quad \textbf{if } r(r_2)(act_2) = yes \wedge r(r_2)(val_2) = 0 \wedge r(r_2)(next_2) = 3 \\
\quad \textbf{then } Out(0, val_2, r_2)(Out(done, o\_r_2\_q_3, r_2)(fs, r)) \\
\quad \textbf{fi} \\
\}
\end{array}
$$

$$
\begin{array}{l}
Produce(r_3) = \lambda(fs, r)\ . \\
\quad \{\textbf{if } r(r_3)(i\_q_2\_r_3) = add \\
\quad \textbf{then } Out(2, next_3, r_3)(Out(yes, act_3, r_3)(Out(1, val_3, r_3)(fs, r))) \\
\quad \textbf{fi}, \\
\quad \textbf{if } r(r_3)(i\_q_4\_r_3) = add \\
\quad \textbf{then } Out(4, next_3, r_3)(Out(yes, act_3, r_3)(Out(1, val_3, r_3)(fs, r))) \\
\quad \textbf{fi}, \\
\quad \textbf{if } r(r_3)(i\_q_2\_r_3) = sub \wedge r(r_3)(val_3) = 0 \\
\quad \textbf{then } Out(0, val_3, r_3)(Out(iszero, o\_r_3\_q_2, r_3)(fs, r)) \\
\quad \textbf{fi}, \\
\quad \textbf{if } r(r_3)(i\_q_4\_r_3) = sub \wedge r(r_3)(val_3) = 0 \\
\quad \textbf{then } Out(0, val_3, r_3)(Out(iszero, o\_r_3\_q_4, r_3)(fs, r)) \\
\quad \textbf{fi}, \\
\quad \textbf{if } r(r_3)(i\_q_2\_r_3) = sub \wedge r(r_3)(val_3) = 1 \\
\quad \textbf{then } Out(2, next_3, r_3)(Out(yes, act_3, r_3)(fs, r)) \\
\quad \textbf{fi}, \\
\quad \textbf{if } r(r_3)(i\_q_4\_r_3) = sub \wedge r(r_3)(val_3) = 1 \\
\quad \textbf{then } Out(4, next_3, r_3)(Out(yes, act_3, r_3)(fs, r))
\end{array}
$$

fi,

if $r(r_3)(act_3) = yes \wedge r(r_3)(val_3) = 1$

then $Out(yes, act_3, r_3)(Out(1, val_3, r_3)(fs, r))$

fi,

if $r(r_3)(act_3) = yes \wedge r(r_3)(val_3) = 0 \wedge r(r_3)(next_3) = 2$

then $Out(0, val_3, r_3)(Out(done, o\_r_3\_q_2, r_3)(fs, r))$

fi,

if $r(r_3)(act_3) = yes \wedge r(r_3)(val_3) = 0 \wedge r(r_3)(next_3) = 4$

then $Out(0, val_3, r_3)(Out(done, o\_r_3\_q_4, r_3)(fs, r))$

fi

}

■

1.4 Summary

In this paper, we have shown that PFF–RDFD's have the computational power of Turing Machines. Therefore, all interesting decidability problems such as reachability, termination, deadlock and liveness properties, and finiteness, that are undecidable for Turing Machines are undecidable for PFF–RDFD's, too.

Future work will have two directions: (i) simulation of FDFD's that make use of persistent flows, stores, infinite domains for flow values, and the facility for testing for empty flows through PFF–RDFD's and (ii) further restrictions on RDFD's. Direction (i) should help to provide a mechanism such that existing FDFD's can be transformed into a basic model. Then, such a model might be used as input to computer software for formal analysis and execution of FDFD's such as the ML interpreter described in [Wah95].

In direction (ii), we hope to find subclasses of RDFD's where some decidability problems can be solved. In particular, we would like to show that some of these subclasses can be simulated by Monogeneous FIFO Petri Nets, Linear FIFO Petri Nets, and Topologically Free Choice FIFO Petri Nets, respectively, and that this simulation is still based on an isomorphism. Then, since this type of a homomorphism preserves some decidability problems ([KM82]), we could immediately apply the results and algorithms known for a subclass of FIFO Petri Nets ([FM82], [MF85], [Fin86], [Rou87], [FC88], [FR88], to mention only a few) to the corresponding subclass of RDFD's.

Acknowledgements

Symanzik's research was partially supported by a German "DAAD–Doktorandenstipendium aus Mitteln des zweiten Hochschulsonderprogramms".

Bibliography

[BB93] J.P. Bansler and K. Bødker. A Reappraisal of Structured Analysis: Design in an Organizational Context. *ACM Transactions on Information Systems*, 11(2):165–193, 1993.

[CB94] D.L. Coleman and A.L. Baker. Synthesizing Structured Analysis and Object–Oriented Specifications. Technical Report 94-04, Iowa State University, Department of Computer Science, 226 Atanasoff Hall, Ames, Iowa 50011, March 1994. Available by anonymous ftp from ftp.cs.iastate.edu or by e-mail from almanac@cs.iastate.edu.

[Col91] D.L. Coleman. *Formalized Structured Analysis Specifications.* PhD Thesis, Iowa State University, Ames, Iowa, 50011, 1991.

[DeM78] T. DeMarco. *Structured Analysis and System Specification.* Yourdon, Inc., New York, New York, 1978.

[FC88] A. Finkel and A. Choquet. FIFO Nets Without Order Deadlock. *Acta Informatica*, 25(1):15–36, 1988.

[Fin86] A. Finkel. *Structuration des Systemes de Transitions — Applications au Controle du Parallelisme par Files FIFO.* These Science, Universite de Paris–Sud, Centre d'Orsay, 1986.

[FM82] A. Finkel and G. Memmi. FIFO Nets: A New Model of Parallel Computation. In A.B. Cremers and H.P. Kriegel, editors, *Lecture Notes in Computer Science Vol. 145: Theoretical Computer Science: 6th GI–Conference, Dortmund, January 1983*, pages 111–121, Springer–Verlag, Berlin, Heidelberg, 1982.

[FR88] A. Finkel and L. Rosier. A Survey on the Decidability Questions for Classes of FIFO Nets. In G. Rozenberg, editor, *Lecture Notes in Computer Science Vol. 340: Advances in Petri Nets 1988*, pages 106–132, Springer–Verlag, Berlin, Heidelberg, 1988.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Har92] D. Harel. Biting the Silver Bullet. *Computer*, 21(1):8–20, January 1992.

[Har96] D. Harel. Executable Object Modeling with Statecharts. In *Proceedings of the 18th International Conference on Software Engineering*, pages 246–257. IEEE Computer Society Press, January 1996.

[Jen80] K. Jensen. A Method to Compare the Descriptive Power of Different Types of Petri Nets. In P. Dembiński, editor, *Lecture Notes in Computer Science Vol. 88: Mathematical Foundations of Computer Science 1980: Proceedings of the 9th Symposium Held in Rydzyna, Poland, September 1980*, pages 348–361, Springer–Verlag, Berlin, Heidelberg, 1980.

[KM82] T. Kasai and R.E. Miller. Homomorphisms between Models of Parallel Computation. *Journal of Computer and System Sciences*, 25:285–331, 1982.

[LWBL96] G.T. Leavens, T. Wahls, A.L. Baker, and K. Lyle. An Operational Semantics of Firing Rules for Structured Analysis Style Data Flow Diagrams. Technical Report 93–28d, Iowa State University, Department of Computer Science, 226 Atanasoff Hall, Ames, Iowa 50011, December 1993, revised, July 1996. Available by anonymous ftp from ftp.cs.iastate.edu or by e–mail from almanac@cs.iastate.edu.

[MF85] G. Memmi and A. Finkel. An Introduction to FIFO Nets — Monogeneous Nets: A Subclass of FIFO Nets. *Theoretical Computer Science*, 35(2–3):191–214, 1985.

[Min67] M.L. Minsky. *Computation: Finite and Infinite Machines.* Prentice–Hall, Inc., Englewood Cliffs, New Jersey, 1967.

[MM81] R. Martin and G. Memmi. Specification and Validation of Sequential Processes Communicating by FIFO Channels. *I.E.E. Conference Publication No. 198: Fourth International Conference on Software Engineering for Telecommunication Switching Systems, Warwick, July 1981*, pages 54–57, 1981.

[Rou87] G. Roucairol. FIFO–Nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Lecture Notes in Computer Science Vol. 254: Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, September 1986*, pages 436–459, Springer–Verlag, Berlin, Heidelberg, 1987.

[SB96] J. Symanzik and A.L. Baker. Non–Atomic Components of Data Flow Diagrams: Stores, Persistent Flows, and Tests for Empty Flows. Technical Report 96–21, Iowa State Uni-

versity, Department of Computer Science, 226 Atanasoff Hall, Ames, Iowa 50011, December 1996. Available by anonymous ftp from ftp.cs.iastate.edu or by e–mail from almanac@cs.iastate.edu.

[TP89] T.H. Tse and L. Pong. Towards a Formal Foundation for DeMarco Data Flow Diagrams. *The Computer Journal*, 32(1):1–12, February 1989.

[VVN81] R. Valk and G. Vidal-Naquet. Petri Nets and Regular Languages. *Journal of Computer and System Sciences*, 23:299–325, 1981.

[Wah95] T. Wahls. *On the Execution of High Level Formal Specifications*. PhD Thesis, Iowa State University, Ames, Iowa, 50011, 1995.

[WBL93] T. Wahls, A.L. Baker, and G.T. Leavens. An Executable Semantics for a Formalized Data Flow Diagram Specification Language. Technical Report 93–27, Iowa State University, Department of Computer Science, 226 Atanasoff Hall, Ames, Iowa 50011, November 1993. Available by anonymous ftp from ftp.cs.iastate.edu or by e–mail from almanac@cs.iastate.edu.

[WM85a] P.T. Ward and S.J. Mellor. *Structured Development for Real–Time Systems*, Volume 1: Introduction and Tools. Yourdon, Inc., New York, New York, 1985.

[WM85b] P.T. Ward and S.J. Mellor. *Structured Development for Real–Time Systems*, Volume 2: Essential Modeling Techniques. Yourdon, Inc., New York, New York, 1985.

[You89] E. Yourdon. *Modern Structured Analysis*. Yourdon Press Computing Series. Prentice–Hall, Englewood Cliffs, New Jersey, 1989.