Non–Atomic Components of Data Flow Diagrams: Stores, Persistent Flows, and Tests for Empty Flows

Jürgen Symanzik and Albert L. Baker

TR #96–21

December 1996

Keywords: Software Specifications, Data Flow Diagrams, Models of Computation.

Department of Computer Science

226 Atanasoff Hall

Iowa State University

Ames, Iowa 50011–1040, USA

1 NON–ATOMIC COMPONENTS OF DATA FLOW DIAGRAMS: STORES, PERSISTENT FLOWS, AND TESTS FOR EMPTY FLOWS

Abstract

It has been shown in [SB96] that a particular subclass of Formalized Data Flow Diagrams (FDFD's) is Turing equivalent. We call this Turing equivalent subclass of FDFD's persistent flow–free Reduced Data Flow Diagrams (PFF–RDFD's). PFF–RDFD's do not contain persistent flows, reference only values whose types have finite domains, and have enabling conditions that contain no tests for empty flows. In addition, FDFD's do not contain (direct) representations of stores. This raises the question whether any of these common features of traditional Data Flow Diagrams elevates the expressive power of FDFD's, or whether the various subclasses have the same expressive power as FDFD's with these features. This paper addresses this issue of whether persistent flows, arbitrary domains, tests for empty flows or stores are essential features with respect to the expressive power of Formalized Data Flow Diagrams.

1.1 Introduction

Traditional Data Flow Diagrams (DFD's) are probably the most widely used specification technique in industry today. They are the cornerstone of the software development methodology commonly referred to as "Structured Analysis" (SA) ([You89]). Their popularity arises from their graphical representation and hierarchical structure, which allows users with non–technical backgrounds to work with them.

One of the drawbacks of traditional DFD's as a specification tool is that they have no rigorous or standard interpretation. In particular, traditional DFD's are usually considered static roadmaps of information flow in systems in which the *bubbles* — data transformers — of DFD's are the cities and *data flows* are the roads. Thus, a DFD can not specify system functionality, i. e., a DFD can not define the I/O behavior of a system. (This would require modelling of car movement in the roadmap analogy.) But defining system functionality is one of the things specifications should do.

Numerous formalizations of DFD's have appeared in the technical literature, e. g., in [DeM78], [WM85a], [WM85b], [Har87], [TP89], [You89], [Har92], and [Har96]. These attempts involve, in part, a more dynamic interpretation of data movement in DFD's. Given this type of rigorous and dynamic semantics, a DFD can serve as a formal specification of system functionality.

The authors use the approach to formalizing DFD's developed originally in [Col91], [CB94], [WBL93] and refined more precisely in [WBL93] and [LWBL96]. These Formalized Data Flow Diagrams (FDFD's) are described more fully in the next section. The formalization is based on defining bubble behavior in terms of *enabling conditions*, which define the pre–state required for a bubble to "do its thing", and *post–conditions*, which define a bubble's outputs in terms of its inputs. The enabling conditions and post–conditions are defined assertionally using *First Order Predicate Calculus* (FOPC) over abstract types, and are refered to as *firing rules*[1].

In other recent work the authors have shown that a restricted class of FDFD's (we call them PFF–RDFD's) is Turing equivalent ([SB96]). The FDFD's used in this proof are those without persistent flows, without stores, and without enabling conditions that check that a flow is empty. Otherwise, we know that the problem of satisfiability in the predicate calculus is unsolvable ([LP81], p. 435). This poses the question whether these non–atomic components are fundamental to FDFD's, in that they might rise the computational power beyond that of Turing Machines, or whether they are simply

[1] Wahls has developed an interpreter for an expressive subset of FOPC assertions over abstract types. The latest reference on this work is [WBL93]. This interpreter can be viewed as an executable semantics for what he refers to as *constructive assertions*. It can also be viewed as a prototype for a CASE tool providing direct execution of abstract model–based specifications.

syntactic features included for expressive convenience. We show in this paper that FDFD's that use these features can be transformed into equivalent PFF–RDFD's that do not contain these features.

Section 1.2 provides an introduction to DFD's, FDFD's, Reduced FDFD's (RDFD's), and persistent flow–free RDFD's (PFF–RDFD's). In Section 1.3 we show that FDFD's with persistent flows, with stores, and with enabling conditions that check for empty flows can all be expressed as equivalent PFF–RDFD's without any of these features. Section 1.4 deals with the nature of the domains of values that can serve as types for values referenced in FDFD's. In the concluding Section 1.5 we stress that our goal is not to eliminate use of these features from FDFD's in actual development environments — they are expressively quite convenient — but to, without loss of generality, use PFF–RDFD's in further analysis of the computational behavior of FDFD's.

1.2 Formalized Data Flow Diagrams

Traditional DFD's are composed from four basic components: *bubbles*, *flows*, *stores*, and *terminators*. Bubbles represent either processes or procedures and are depicted as circles. In a more abstract sense, bubbles are viewed as data transformers.

Flows represent the paths over which data may travel (i. e., the roads, not the cars, in the roadmap analogy). They can represent data flow from terminators to bubbles, bubbles to bubbles, bubbles to stores, stores to bubbles, or bubbles to terminators. If we view bubbles, stores and terminators as nodes, then a DFD is simply a directed graph in which the flows are the arcs. From the perspective of a bubble, flows into the bubble are called *inflows* and flows out of the bubble are called *outflows*.

Terminators are the sources of input to and destinations of output from the system being specified. They are depicted as rectangles and can be viewed as processes that are not part of the system being specified.

In traditional DFD's, stores are viewed as data at rest (whatever that means). Stores are often just thinly veiled abstractions for files. They are often depicted as rectangles with open sides.

1.2.1 The Syntax of FDFD's

Since [Col91] provides a formal syntax and [WBL93] provides a semantics for FDFD's, we refer to the language for expressing FDFD's as DFD–SPECS. The presentation in the rest of this section is less formal than in the cited references and we continue to refer to FDFD's (rather than DFD–SPECS).

An FDFD consists of a directed graph and an associated textual part. As in traditional DFD's, the nodes of the graph are the bubbles, and the arcs are the flows.

Stores in traditional DFD's are represented with one or more inflows, representing the flow of data values to be stored, and one or more outflows, representing the flow of data values to be retrieved. This has always seemed a rather informal view of persistent repositories of data. Even if stores are just modelled as abstract data types, then a bubble adding a data value to a store would need to designate which constructor operation of the ADT is to be used. Similarly, a bubble obtaining a data value from a store would need to designate a particular selector operation.

Coleman suggests modelling stores as persistent flows with multiple originating bubbles, representing bubbles adding data values to a store, and multiple destination bubbles, representing bubbles obtaining data values from a store [Col91]. This perspective on stores as flows with multiple origin and destination bubbles is adopted in the syntax of FDFD's.

Terminators are depicted simply as bubbles in which either they have no inflows, representing the data sources, or in which they have no outflows, representing the data sinks. These bubbles are at most only partially specified.

Each bubble in the directed graph portion of an instance of an FDFD has a unique name label. Each flow is labelled with its name and type. Dashed arcs are used for *persistent* flows, while solid arcs are used for *consumable* flows. (Persistent and consumable flows are described in Subsection 1.2.2.)

The textual part of an FDFD consists of the data dictionary of types and bubble firing rules. At this specification level, these types are viewed as abstract types. They are specified using a formal, model–based approach, similar to that of [GHG+93] and [Jon86]. The firing rules defining the behavior of bubbles are expressed as assertions over the abstract types. The language for writing assertions is described in the following paragraphs and extended BNF grammar[2].

textual-part ::= [*data-dictionary*] *process**

The data dictionary defines the abstract types and abstract functions over these types used in the firing rules. The notation *type-expr*$^{*|}$ refers to union types, i. e., `type intOrReal = int | real;`.

data-dictionary ::= `Data Dictionary :` *type-decl*$^{*;}$ [`;`] *abstract-function*$^{*;}$ [`;`]

type-decl ::= `type` *var-name* `=` *type-expr*

type-expr ::= `int` | `real` | `bool` | `string` | `signal` | `set of` *type-expr* | *type-expr*$^{*|}$

`sequence of` *type-expr* | `tuple of (`*param-decl*$^{*;}$`)` | *type-name*

Abstract functions just serve to help modularize the specifications. An abstract function that defines type bool is really just a predicate. However, the model–based specification language does support the definition of abstract functions that define other abstract types.

[2] In this grammar, optional parts are enclosed in square brackets [ ], and the notation *expr*$^{*;}$ means a ; separated list of zero or more *expr*s.

abstract-function ::=**define** *absfun-name*(*param-decl**,) **as** *type-expr*

such that *FOPC-expr*

param-decl ::= *var-name* : *type-expr*

Each bubble is described by its name, initial state, and set of firing rules. The initial state specifies the initial values on the bubble's outflows. For flows with multiple source bubbles, the values specified must be consistent with respect to type. Each firing rule contains an enabling condition and post–condition, as discussed previously. In the enabling condition and initial state, the assertion $^{+}$*flow-name* is true exactly when at least one value is present on flow *flow-name*, while $^{-}$*flow-name* is true when no value is present. An omitted pre–condition is equivalent to just **true**.

process ::= **Process** *bubble-name* **:** [*initial-state*] *rule*$^{*;}$[;]

initial-state ::= **initially** *flow-enabled-list* [$\wedge$ *FOPC-expr*]

rule ::= **enabled when** *enabling-condition* **ensures** *post-condition*

enabling-condition ::= **true** | *flow-enabled-list* [$\wedge$ *FOPC-expr*]

flow-enabled-list ::= [*flow-enabled-list* $\wedge$] *flow-enabled*

flow-enabled ::= $^{+}$*flow-name* | $^{-}$*flow-name*

post-condition ::= *FOPC-expr*

The First Order Predicate Calculus used in FDFS's is augmented with operations on the built–in types, e. g., set. Unprimed flow names refer to the values on inflows, while primed flow names (') refer to outflow values. For each field of a tuple, FDFD's provide a function with the same name to extract that field from the tuple. The symbol **-** is used for both arithmetic subtraction and set difference, and || denotes concatenation of sequences. The **index** function provides array–like indexing into sequences, **header** returns all of its argument sequence except the last element, and **trailer** returns all of its argument sequence except the first element.

FOPC-expr ::= **true** | **false** | **not** *FOPC-expr* | *FOPC-expr* $\wedge$ *FOPC-expr* |

FOPC-expr $\vee$ *FOPC-expr* | *FOPC-expr* $\Rightarrow$ *FOPC-expr* |

$\forall$ *var-name* **:** *type-expr* **[***FOPC-expr***]** |

$\exists$ *var-name* **:** *type-expr* **[***FOPC-expr***]** |

{*FOPC-expr* | *FOPC-expr***}** | (*FOPC-expr*)**:** *type-expr* |

int-literal | *real-literal* | *string-literal* | *bool-literal* | *var-name* |

flow-name | *flow-name'* | *absfun-name*(*FOPC-expr**,) |

unary-op(*FOPC-expr*) | *FOPC-expr binary-op FOPC-expr* |

`{`*FOPC-expr**,`}` | `<`*FOPC-expr**,`>` | `(`*FOPC-expr**,`)` |

`index(`*FOPC-expr*, *FOPC-expr*`)`

unary-op ::= *field-name* | `size` | `first` | `header` | `last` | `trailer` | `length`

binary-op ::= $+$ | $-$ | $*$ | $/$ | $\%$ | $\cup$ | $\cap$ | $\mid\mid$ | $=$ | $<$ | $\leq$ | $>$ | $\geq$ | $\in$ | $\subset$ | $\subseteq$ | $\supset$ | $\supseteq$

1.2.2 An Informal Semantics of FDFD's

This informal description of FDFD's semantics is based on the previously referenced works ([CB94] and [LWBL96]) and on the interpreter developed by Wahls. The key concept in providing a meaning of FDFD's that allow them to serve as formal functional specifications is that of *firing* a bubble. Succinctly, firing is the process by which a bubble reads its values from its inflows and produces values on its outflows.

Bubbles fire in two steps. In the first step, a bubble reads values from its inflows, and in the second step, it writes values to its outflows. We say a bubble is *working* when it has read its inflows, but not yet produced values on its outflows. A bubble is *idle* otherwise. We treat the transitions between these states as atomic.

The effect of reading values from a flow depends on the type of the flow. When a bubble reads from a consumable flow, the value read is removed from the flow. Thus, consumable flows can be viewed as first–in, first–out unbounded queues of values, where each value is of the type associated with the flow. Reading the value of a persistent flow does not affect the flow value. When a bubble "outputs" a value to a consumable flow, that value is just appended to the back of the queue of values. Writing to a persistent flow overwrites any previous value. Thus, a persistent flow can be viewed as a variable shared between a process that writes the variable and a process that reads the variable.

Bubble firing occurs as follows. Initially, all bubbles are idle. The flows may have initial values that are specified as part of the initial state.

(i) Find the set of bubbles that may fire. This includes all bubbles in the working state, and any bubble in the idle state that has values on its inflows satisfying the enabling condition of at least one of its firing rules.

(ii) Choose one of these bubbles to fire.

(iii) Fire the bubble:

- If the bubble is *idle*:

(a) Choose one of the bubble's rules whose enabling condition is satisfied by the inflow values.

(b) Read the values referenced by this rule from the inflows. For consumable flows, remove the value. Otherwise, do not change the flow.

(c) Change the state of the bubble from *idle* to *working*.

- If the bubble is *working*:

(a) Produce values onto the outflows. These values are defined by the post–condition of the rule chosen when the bubble changed to the working state. For consumable flows, the value is enqueued. For persistent flows, the new value overwrites the flow's contents.

(b) Change the state of the bubble from *working* to *idle*.

(iv) Repeat the above steps until the set of bubbles allowed to fire in step one is empty.

1.2.3 Restricted Classes of FDFD's

As mentioned earlier, the authors have shown that a particular restricted class of FDFD's is Turing equivalent [SB96]. In this section we define this class of FDFD's.

The first restriction is on the nature of enabling conditions. The enabling condition must do more than test for the presence of a value on a flow. For each such presence test, it must also contain an assertion limiting the value on that flow.

Definition (1.2.3.1): An enabling condition with no tests for the absence of a value on a flow (i. e., no boolean expressions ^{-}f) and in which every boolean expression ^{+}f has associated with it an assertion further bounding the value of f to a single value is called a *normal form enabling condition.* If f is a persistent flow, the ^{+}f can be omitted and only the assertion bounding the value of f is required. ■

The next restriction applies to entire FDFD's. It limits the domains of abstract types and requires normal form enabling conditions.

Definition (1.2.3.2): A *Reduced Formalized Data Flow Digram* (RDFD) is an FDFD in which (i) every abstract type modelled and referenced in the FDFD has a finite domain, (ii) sequences and tuples are restricted to a finite maximum length and (iii) every enabling condition is a normal form enabling condition. ■

Because of the finite domain and the length restriction of sequences and tuples every assertion in predicate calculus becomes solvable since the two quantifiers $\forall$ and $\exists$ are bound to a finite number of objects available. We want to stress again that, even though we do not allow sequences of arbitrary (or infinite) length as a single object, we do not prevent the production of infinite many objects of a fixed length on any of the flows.

Finally, we restrict RDFD's to preclude persistent flows:

Definition (1.2.3.3): An RDFD that does not have any persistent flows is called a *persistent flow–free Reduced (Formalized) Data Flow Diagram* (PFF–RDFD). ■

It is PFF–RDFD's that is shown to be Turing equivalent in [SB96]. In the following three sections we argue that these restrictions can be made without loss of generality.

1.3 Transformation of FDFD's with Non–Atomic Components into PFF–RDFD's

In this section we show how to replace the test for empty flows (Example 1.3.1), persistent flows (Example 1.3.2), and stores (Example 1.3.3) by features provided by PFF–RDFD's. These examples could be easily extended to more complex situations. Obviously, another feature of FDFD's, i. e., infinite domains for flow values, can be resolved, for example, by using the unary or binary representation of objects. This type of encoding is called a *Gödel numbering*, after the logician Kurt Gödel. Thus, any object can be represented as a finite sequence of 0's and 1's.

Example (1.3.1): This example contains two bubbles. One of them (P) can always fire, producing 1's on its outflow f. The other bubble (C) will produce the value 0 on its outflow *out* as long as its inflow f is empty and the value 1 if it is not empty. The output possible on flow *out* is $\{0,1\}^*$.

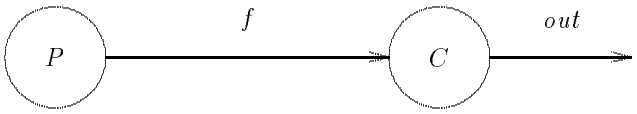

Figure 1.1: Example with Test for Empty Flow.

The assertions for the FDFD shown in Figure 1.1 using the test for empty flows (^{-}f) are specified as follows:

Process P:

$true$:

$\models f' = 1$

Process C:

^{-}f :

$\models out' = 0;$

$^{+}f \wedge f = 1$:

$\models out' = 1$

Initial State:

$^{-}f \wedge ^{-} out$

To replace the ^{-}f in the enabling condition of bubble C, we use a controller bubble Z in our PFF–RDFD (see Figure 1.2) to determine whether flow f is empty or not. When bubble P produces a value on f, it also produces a signal *fproduced* on $Pdone$ to inform Z on the existence of a new value on f. However, the value on f will not immediately be available for C since Z continues to send *fisempty* on flow $Cenab$ as long as $fcount = 0$ holds. Instead, Z has to consume the signal on $Pdone$ first and increment the counter $fcount$. Then, since $fcount \geq 1$ holds, the next value on $Cenab$ will be *fnotempty*, allowing C to consume the value on f. The output possible on flow out is $\{0,1\}^*$ as in the original example.

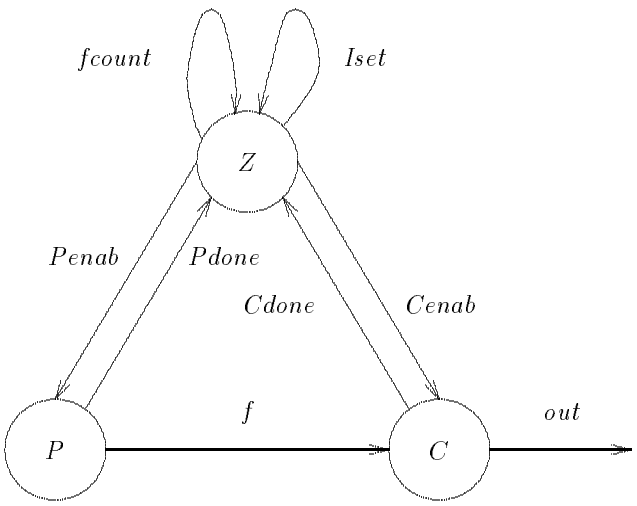

Figure 1.2: Example without Test for Empty Flow.

For the ease of our description we informally use the unbounded integer variable *fcount*. Using the unary or binary representation of this variable would solve the problem of an infinite number of integer objects but it would also extend the coding of this example which is not desired.

Initially, *fcount* contains 0, i. e., no value is available on flow f. Flow *Iset* indicates which bubbles are idle, i. e., initially P and C. Possible values on the flows are PC, P, C, and O (which indicates neither P nor C is idle) on *Iset*, *fisempty* and *fnotempty* on *Cenab*, *fconsumed* and *nofconsumed* on *Cdone*, *go* on *Penab*, and *fproduced* on *Pdone*.

Process Z:

$^+ Iset \wedge Iset = PC \wedge^+ fcount \wedge fcount = 0 :$

$\models Penab' = go \wedge Iset' = C \wedge fcount' = fcount$

$\square\, Cenab' = fisempty \wedge Iset' = P \wedge fcount' = fcount;$

$^+ Iset \wedge Iset = PC \wedge^+ fcount \wedge fcount \geq 1 :$

$\models Penab' = go \wedge Iset' = C \wedge fcount' = fcount$

$\square\, Cenab' = fnotempty \wedge Iset' = P \wedge fcount' = fcount;$

$^+ Iset \wedge Iset = P \wedge^+ fcount \wedge fcount \geq 0 :$

$\models Penab' = go \wedge Iset' = O \wedge fcount' = fcount;$

$^+ Iset \wedge Iset = C \wedge^+ fcount \wedge fcount = 0 :$

$\models Cenab' = fisempty \wedge Iset' = O \wedge fcount' = fcount;$

$^+ Iset \wedge Iset = C \wedge^+ fcount \wedge fcount \geq 1 :$

$\models Cenab' = fnotempty \wedge Iset' = O \wedge fcount' = fcount;$

$^+ Pdone \wedge Pdone = fproduced \wedge^+ Iset \wedge Iset = O \wedge^+ fcount \wedge fcount \geq 0 :$

$\models Iset' = P \wedge fcount' = fcount + 1;$

$^+ Pdone \wedge Pdone = fproduced \wedge^+ Iset \wedge Iset = C \wedge^+ fcount \wedge fcount \geq 0 :$

$\models Iset' = PC \wedge fcount' = fcount + 1;$

$^+ Cdone \wedge Cdone = nofconsumed \wedge^+ Iset \wedge Iset = O \wedge^+ fcount \wedge fcount \geq 0 :$

$\models Iset' = C \wedge fcount' = fcount;$

$^+ Cdone \wedge Cdone = nofconsumed \wedge^+ Iset \wedge Iset = P \wedge^+ fcount \wedge fcount \geq 0 :$

$\models Iset' = PC \wedge fcount' = fcount;$

$^+ Cdone \wedge Cdone = fconsumed \wedge^+ Iset \wedge Iset = O \wedge^+ fcount \wedge fcount \geq 1 :$

$\models Iset' = C \wedge fcount' = fcount - 1;$

$^+ Cdone \wedge Cdone = fconsumed \wedge^+ Iset \wedge Iset = P \wedge^+ fcount \wedge fcount \geq 1 :$

$\models Iset' = PC \wedge fcount' = fcount - 1$

Process P:

$^{+}Penab \wedge Penab = go$:

$\models f' = 1 \wedge Pdone' = fproduced$

Process C:

$^{+}Cenab \wedge Cenab = fisempty$:

$\models out' = 0 \wedge Cdone' = nofconsumed$;

$^{+}Cenab \wedge Cenab = fnotempty \wedge^{+} f \wedge f = 1$:

$\models out' = 1 \wedge Cdone' = fconsumed$

Initial State:

$fcount = 0 \wedge Iset = PC \wedge^{-} f \wedge^{-} out \wedge^{-} Penab \wedge^{-} Pdone \wedge^{-} Cenab \wedge^{-} Cdone$ ■

Example (1.3.2): As in the previous example, this example also contains two bubbles. Here bubble A can always fire, producing 0's and 1's on its persistent outflow f. Bubble B will produce the value 0 on its outflow out if it reads a 0 on its inflow f and the value 1 if it reads a 1. The output possible on flow out is $\{0,1\}^*$.

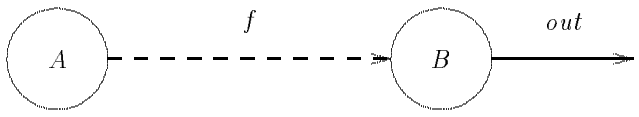

Figure 1.3: Example with Persistent Flow.

The assertions for the FDFD shown in Figure 1.3 are specified as follows:

Process A:

$true$:

$\models f' = 0$

$\Box f' = 1$

Process B:

$f = 0$:

$\models out' = 0$;

$f = 1$:

$\models out' = 1$

Initial State:

$(f = 0 \vee f = 1) \wedge^{-} out$

This time, we use the controller bubble Z in our PFF–RDFD (see Figure 1.4) to inform bubble B on the latest value that has been generated by bubble A. However, instead of writing this new value on a persistent flow, A forwards this new value to Z. Once Z has read this value from its inflow $Adone$, it will be stored on flow $fval$. The next value on flow $Benab$ will be the value currently stored in $fval$. The output possible on flow out is $\{0,1\}^*$ as in the original example.

We are using the somewhat sloppy notation $0/1$ where we mean that either the value 0 or the value 1 is available. It should be obvious how to extend this notation such that the mappings are in accordance with our definition of a PFF–RDFD.

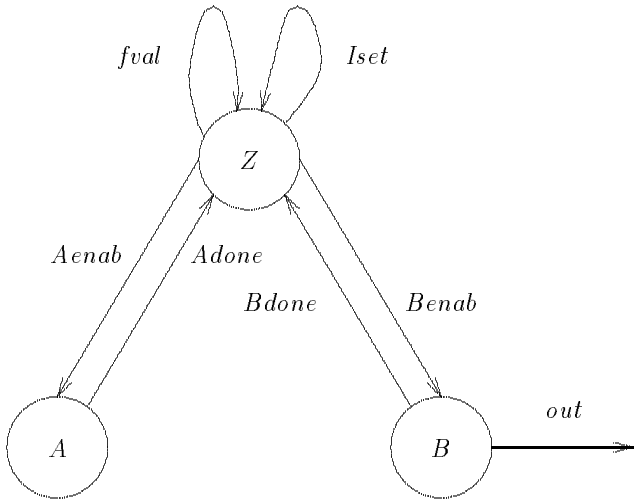

Figure 1.4: Example without Persistent Flow.

Initially, $fval$ contains either a 0 or a 1. Flow $Iset$ indicates which bubbles are idle, i. e., initially A and B. Possible values on the flows are AB, A, B, and O (which indicates neither A nor B is idle) on $Iset$, 0 and 1 on $Benab$, $done$ on $Bdone$, go on $Aenab$, and 0 and 1 on $Adone$.

Process Z:

$^{+}Iset \wedge Iset = AB \wedge^{+} fval \wedge fval = 0/1 :$

$\models Aenab' = go \wedge Iset' = B \wedge fval' = fval$

$\Box Benab' = fval \wedge Iset' = A \wedge fval' = fval;$

$^{+}Iset \wedge Iset = A \wedge^{+} fval \wedge fval = 0/1 :$

$\models Aenab' = go \wedge Iset' = O \wedge fval' = fval;$

$^{+}Iset \wedge Iset = B \wedge^{+} fval \wedge fval = 0/1 :$

$\models Benab' = fval \wedge Iset' = O \wedge fval' = fval;$

$^{+}Adone \wedge Adone = 0 \wedge^{+} Iset \wedge Iset = O \wedge^{+} fval \wedge fval = 0/1 :$

$\models Iset' = A \wedge fval' = 0;$

$^{+}Adone \wedge Adone = 1 \wedge^{+} Iset \wedge Iset = O \wedge^{+} fval \wedge fval = 0/1 :$

$\models Iset' = A \wedge fval' = 1;$

$^{+}Adone \wedge Adone = 0 \wedge^{+} Iset \wedge Iset = B \wedge^{+} fval \wedge fval = 0/1 :$

$\models Iset' = AB \wedge fval' = 0;$

$^{+}Adone \wedge Adone = 1 \wedge^{+} Iset \wedge Iset = B \wedge^{+} fval \wedge fval = 0/1 :$

$\models Iset' = AB \wedge fval' = 1;$

$^{+}Bdone \wedge Bdone = done \wedge^{+} Iset \wedge Iset = O \wedge^{+} fval \wedge fval = 0/1 :$

$\models Iset' = B \wedge fval' = fval;$

$^{+}Bdone \wedge Bdone = done \wedge^{+} Iset \wedge Iset = A \wedge^{+} fval \wedge fval = 0/1 :$

$\models Iset' = AB \wedge fval' = fval$

Process A:

$^{+}Aenab \wedge Aenab = go :$

$\models Adone' = 0$

$\square Adone' = 1$

Process B:

$^{+}Benab \wedge Benab = 0 :$

$\models out' = 0 \wedge Bdone' = done;$

$^{+}Benab \wedge Benab = 1 :$

$\models out' = 1 \wedge Bdone' = done$

Initial State:

$(fval = 0 \vee fval = 1) \wedge Iset = AB \wedge^{-} out \wedge^{-} Aenab \wedge^{-} Adone \wedge^{-} Benab \wedge^{-} Bdone$ ■

Stores are a common feature of traditional DFD's. They are usually used to represent persistent data, often with the intended implementation using files. Thus stores are usually represented with inflows representing data to be added to the store and outflows with data retrieved from the store. Since there has been no formalism for representing different "constructor" operations to add to a store and no formalism for representing different "selector" or "query" operations for getting data from a

store, stores have not been included in FDFD's in [LWBL96], although a possible extension has been mentioned.

However, the question of whether stores, i. e., the usual way stores are used in traditional DFD's, can be modeled using just the features of PFF–RDFD's is pertinent.

Example (1.3.3): This example demonstrates how to replace the common notion of stores as used in DFD's. Here, we have two bubbles A and B that both read from and write to the same store $Store$. Note that, due to the two phase firing semantics of FDFD's, the value of a flow is read in step 1 while a new value of a flow is written in step 2. For example, when A issues a write command (the value 0), any number of reads from B may return the old value. Even a write from B (the value 1) started later than A's write may be completed earlier than A's write, leaving the value 0 as the final value.

Figure 1.5: Example with Store.

The assertions for the FDFD shown in Figure 1.5 are specified as follows:

Process A:

^{-}SA :

$\models AS' = 0;$

$SA = 0$:

$\models AS' = 0$

$\Box\, out'_A = 0;$

$SA = 1$:

$\models AS' = 0$

$\Box\, out'_A = 1$

Process B:

^{-}SB :

$\models BS' = 1;$

$SB = 0$:

$\models BS' = 1$

$\Box\, out'_B = 0;$

$SB = 1$:

$\models BS' = 1$

$\Box\, out'_B = 1$

Initial State:

$^{-}SA \wedge^{-} SB \wedge^{-} AS \wedge^{-} BS \wedge^{-} out_A \wedge^{-} out_B$

Once again, we make use of a controller bubble Z in our PFF–RDFD (see Figure 1.6). This time, it is used to guarantee that a write does not modify the stored value before all reads issued prior to or during the write have finished returning the old value. If one bubble issues a write while the other bubble's write has not yet finished, the order in which their values are stored is determined nondeterministically.

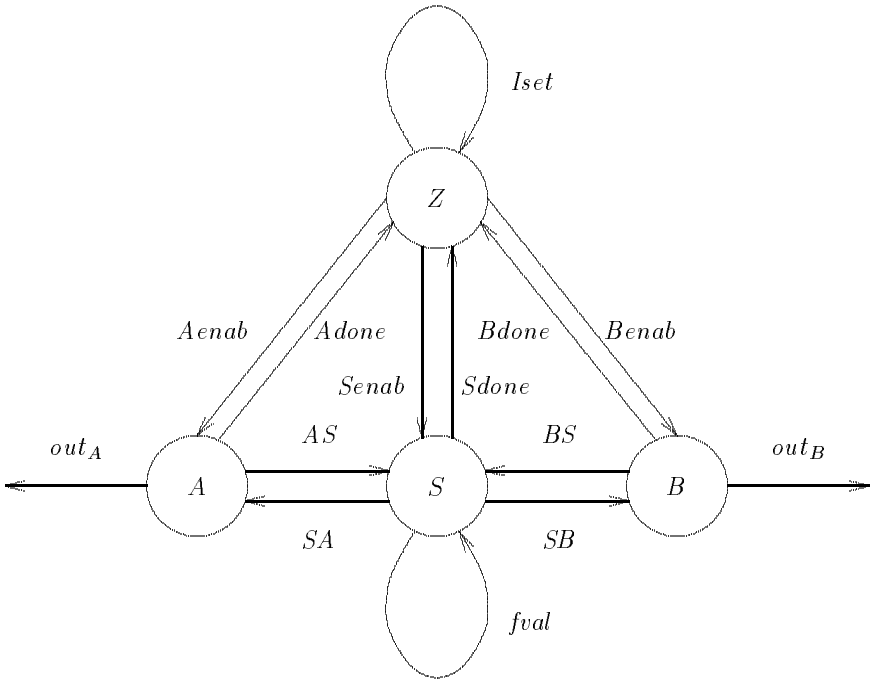

Figure 1.6: Example without Store.

Initially, *fval* contains either a 0 or a 1. Flow *Iset* indicates which bubbles are idle, i. e., initially A, B, and S. Possible values on the flows are ABS, AS, BS, A, B, S, and O (which indicates neither A nor B nor S is idle) on *Iset*, *request* and *write0* on AS, *request* and *write1* on BS, 0 and 1 on SA, on SB, and on *fval*, *go* on *Aenab* and on *Benab*, *fread* and *fproduced* on *Adone* and on *Bdone*, *Aproduced* and *Bproduced* on *Senab*, and *Awritten* and *Bwritten* on *Sdone*.

Process Z:

$^{+}Iset \wedge Iset = ABS$:

$\models Aenab' = go \wedge Iset' = BS$

$\Box Benab' = go \wedge Iset' = AS$;

$^{+}Iset \wedge Iset = AS$:

$\models Aenab' = go \wedge Iset' = S$;

$^{+}Iset \wedge Iset = BS$:

$\models Benab' = go \wedge Iset' = S$;

$^{+}Adone \wedge Adone = fread \wedge^{+} Iset \wedge Iset = S$:

$\models Iset' = AS$;

$^{+}Adone \wedge Adone = fread \wedge^{+} Iset \wedge Iset = BS$:

$\models Iset' = ABS$;

$^{+}Adone \wedge Adone = fproduced \wedge^{+} Iset \wedge Iset = S$:

$\models Senab' = Aproduced \wedge Iset' = O$;

$^{+}Adone \wedge Adone = fproduced \wedge^{+} Iset \wedge Iset = BS$:

$\models Senab' = Aproduced \wedge Iset' = B$;

$^{+}Bdone \wedge Bdone = fread \wedge^{+} Iset \wedge Iset = S$:

$\models Iset' = BS$;

$^{+}Bdone \wedge Bdone = fread \wedge^{+} Iset \wedge Iset = AS$:

$\models Iset' = ABS$;

$^{+}Bdone \wedge Bdone = fproduced \wedge^{+} Iset \wedge Iset = S$:

$\models Senab' = Bproduced \wedge Iset' = O$;

$^{+}Bdone \wedge Bdone = fproduced \wedge^{+} Iset \wedge Iset = AS$:

$\models Senab' = Bproduced \wedge Iset' = A$;

$^{+}Sdone \wedge Sdone = Awritten \wedge^{+} Iset \wedge Iset = O$:

$\models Iset' = AS$;

$^{+}Sdone \wedge Sdone = Awritten \wedge^{+} Iset \wedge Iset = B$:

$\models Iset' = ABS$;

$^{+}Sdone \wedge Sdone = Bwritten \wedge^{+} Iset \wedge Iset = O$:

$\models Iset' = BS$;

$^{+}Sdone \wedge Sdone = Bwritten \wedge^{+} Iset \wedge Iset = A$:

$\models Iset' = ABS$

Process A:

$^{+}Aenab \wedge Aenab = go$:

$\models AS' = request$

$\Box AS' = write0 \wedge Adone' = fproduced;$

$^{+}SA \wedge SA = 0/1$:

$\models out'_A = SA \wedge Adone' = fread$

Process B:

$^{+}Benab \wedge Benab = go$:

$\models BS' = request$

$\Box BS' = write1 \wedge Bdone' = fproduced;$

$^{+}SB \wedge SB = 0/1$:

$\models out'_B = SB \wedge Bdone' = fread$

Process S:

$^{+}AS \wedge AS = request \wedge^{+} fval \wedge fval = 0/1$:

$\models SA' = fval \wedge fval' = fval;$

$^{+}AS \wedge AS = write0 \wedge^{+} fval \wedge fval = 0/1 \wedge Senab = Aproduced$:

$\models fval' = 0 \wedge Sdone' = Awritten;$

$^{+}BS \wedge BS = request \wedge^{+} fval \wedge fval = 0/1$:

$\models SB' = fval \wedge fval' = fval;$

$^{+}BS \wedge BS = write1 \wedge^{+} fval \wedge fval = 0/1 \wedge Senab = Bproduced$:

$\models fval' = 1 \wedge Sdone' = Bwritten$

Initial State:

$(fval = 0 \vee fval = 1) \wedge Iset = ABS \wedge^{-} out_A \wedge^{-} out_B$

$\wedge^{-} Aenab \wedge^{-} Adone \wedge^{-} Benab \wedge^{-} Bdone \wedge^{-} Senab \wedge^{-} Sdone \wedge^{-} SA \wedge^{-} SB \wedge^{-} AS \wedge^{-} BS$ ■

1.4 Transformation of FDFD's with Infinite Domains into PFF–RDFD's

While in the previous section we have considered non–atomic components of FDFD's that are of particular interest for the Structured Analysis, we will now deal with a more theoretical issue, that is, infinite domains. We have mentioned several times that we have to restrict every abstract type to a finite domain and do not allow sequences, tupels, and sets of arbitrary (or infinite) length. The reasons for this restriction follow in the next four subsections.

1.4.1 Quantifiers over Unbounded Sets

Consider the case where the enabling condition reads as

$$^{+}f \wedge \exists x_1 : \texttt{int} \ldots \exists x_n : \texttt{int}\,(P(x_1, \ldots, x_n, f) = 0) \models C_{true}$$

where flow f is of type `int`, and P is a given polynomial of degree $n+1$. The above condition represents a diophantine equation. The problem, whether there exists a procedure which in a finite number of steps enables one to determine whether or not a given diophantine equation has an integer solution is known as Hilbert's 10th problem. It has been shown ([Mat70]) that this problem is undecidable. Therefore, we can express something as a condition for an FDFD, but we are not capable to provide an algorithm that allows us to decide whether this condition holds or does not hold.

However, assume we define `bounded_int = {minint, ..., maxint}` and redefine the enabling condition as

$$^{+}f \wedge \exists x_1 : \texttt{bounded\_int} \ldots \exists x_n : \texttt{bounded\_int}\,(P(x_1, \ldots, x_n, f) = 0) \models C_{true}$$

where flow f is of type `bounded_int`, too. Now this condition becomes decidable for any possible value on flow f since there are only finite many cases that have to be considered. Even more, we could reduce this condition to our normal form enabling condition when designing the model, i. e.,

$$^{+}f \wedge f = \texttt{minint} \models C_{true}$$
$$^{+}f \wedge f = \texttt{minint} + 1 \models C_{true}$$
$$^{+}f \wedge f = \texttt{minint} + 2 \models C_{true}$$
$$\vdots$$
$$^{+}f \wedge f = \texttt{maxint} \models C_{true}$$

where only those cases are listed that result in *true* in the original condition.

1.4.2 Infinite Domains

Assume we have a flow f of type `unsigned int` and only distinguish among the values $0, \ldots, n$:

$$^{+}f \wedge f = 0 \models C_0$$
$$^{+}f \wedge f = 1 \models C_1$$
$$\vdots$$
$$^{+}f \wedge f = n \models C_n$$
$$^{+}f \wedge f > n \models C_{>n}$$

As we have mentioned several times, we can use the unary representation of (unsigned) integers, where the sequence of values $(\underbrace{1, \ldots, 1}_{n}, 0)$ represents n. With the help of an additional flow *last* (initialized

with "0") , we can redesign the previous conditions:

$$^{+}f \wedge f = 0 \wedge^{+} last \wedge last = \text{“0”} \models C_0$$

$$^{+}f \wedge f = 1 \wedge^{+} last \wedge last = \text{“0”} \models last' = \text{“1”}$$

$$^{+}f \wedge f = 0 \wedge^{+} last \wedge last = \text{“1”} \models C_1$$

$$^{+}f \wedge f = 1 \wedge^{+} last \wedge last = \text{“1”} \models last' = \text{“2”}$$

$$\vdots$$

$$^{+}f \wedge f = 0 \wedge^{+} last \wedge last = \text{“n”} \models C_n$$

$$^{+}f \wedge f = 1 \wedge^{+} last \wedge last = \text{“n”} \models last' = \text{“>n”}$$

$$^{+}f \wedge f = 0 \wedge^{+} last \wedge last = \text{“>n”} \models C_{>n}$$

$$^{+}f \wedge f = 1 \wedge^{+} last \wedge last = \text{“>n”} \models last' = \text{“>n”}$$

Now the domain, i. e., the objects that are used on all flows, is finite, i. e., the set $\{0, 1\} \cup \{$ “0”, “1”, . . . , “n”, “>n”$\}$.

The idea to use the unary or binary representation of objects is used throughout the theoretical literature in computer science and this type of encoding is called a *Gödel numbering*, after the logician Kurt Gödel. Of course, we can use the Gödel numbering not only to distinguish among different cases as seen above, but also to do calculus directly based on this representation.

1.4.3 Unbounded Sequences, Types, and Sets

Assume the length of a sequence can grow beyond any bound during the execution of the FDFD, however it remains finite at any time. Instead of sending a single object of type `sequence`, we can remodel our FDFD such that each element of the sequence is written as a single object to the unbounded FIFO flow and an additional delimiter is used to indicate the end of the sequence. The same mechanism can be used for types and sets.

1.4.4 Infinite Sequences, Types, and Sets

Assume a flow f of type `set of int` contains a single object `int`, i. e., the infinite many values of the set integer. Consider a bubble that reads its input from flow f and produces some output on flow *out* is specified as follows:

$$^{+}f \wedge f = \texttt{int} \models out' = \{i + 1 \mid i \in f\}$$

Hence, this bubble is intended to produce the successor of each of the values of its input set. What is the semantics underlying the above construct? In an abstract world, we may assume that we can yield the successors of an infinte set in finite time. However, it is more realistic to assume that we need finite

time (> 0) to obtain the successor of each element. Thus, we need infinite time to yield the successors of an infinite set. Therefore, we should assume that this transition once enabled never terminates.

This behavior to spend infinite time on determing the successors can be remodeled the following way: One bubble produces the infinite set `int` on flow f using the unary representation introduced in Subsection 1.4.2. Another bubble consumes the head element from flow f and produces the output on flow *out*. Both bubbles will alternate between *idle* and *working*, but none of them is expected to reach a state where it finally remains *idle*. Thus, there exists an infinite firing sequence that only uses these two bubbles. Also, another bubble b that makes use of flow *out* as its input can be remodeled to proceed only if it reads an additional delimiter on *out*. However, this delimiter will never be send, so b makes no overall progress (except reading from *out* and eventually storing these values on a flow with b as source and destination). The same mechanism can be used for sequences and types.

1.5 Summary

In this paper, we have shown how to remodel FDFD's into PFF–RDFD's. We have given examples where the test for empty flows (Example 1.3.1), persistent flows (Example 1.3.2), and stores (Example 1.3.3) have been replaced by features provided by PFF–RDFD's. We have also shown why the use of another feature of FDFD's, i. e., infinite domains, has to be restricted, and how to do so using the unary or binary representation of objects.

Even though we have not provided a general algorithm that transfers any given FDFD into a PFF–RDFD, it should be obvious how our idea could be easily extended to more complex situations. In particular, we know that PFF–RDFD's have the computational power of Turing Machines ([SB96]). From the given examples, it should be obvious that any additional non–atomic component only adds to the expressive power of the model but does not allow to model features unavailable for the basic PFF–RDFD and Turing Machine, respectively.

Acknowledgements

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

Bibliography

[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.

[GHG+93] J.V. Guttag, J.J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer–Verlag, New York, 1993.

[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.

[Jon86] C.B. Jones. *Systematic Software Development Using VDM.* International Series in Computer Science. Prentice–Hall, Englewood Cliffs, New Jersey, 1986.

[LP81] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation.* Prentice–Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[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.

[Mat70] J.V. Matijasevic. Enumerable Sets are Diophantine. *Soviet Mathematics, Doklady*, 11(2):354–358, 1970.

[SB96] J. Symanzik and A.L. Baker. Formalized Data Flow Diagrams and Their Relation to Other Computational Models. Technical Report 96–20, Iowa State University, 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.

[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.