Formal Semantics for Structured Analysis Style Data Flow Diagram Specification Languages

Gary T. Leavens
Iowa State University

Tim Wahls
Iowa State University

Albert L. Baker
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports
Part of the Systems Architecture Commons, and the Theory and Algorithms Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/123

This Article is brought to you for free and open access by the Computer Science at Iowa State University Digital Repository. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Formal Semantics for Structured Analysis Style Data Flow Diagram Specification Languages

Abstract
Using operational semantic techniques, we present a formal semantics for an extended variant of structured analysis style data flow diagrams. This semantics is intended to serve as a semantic foundation for many different specification languages that specify concurrent systems using a graphical notation similar to data flow diagrams. Besides allowing one to specify how information is processed, it allows one to specify the dynamic behavior of a concurrent system. We discuss various semantic issues, including the need for a two-step firing rule and how the semantics supports the notion of refinement.

Keywords
structured analysis, data flow diagram, semantics of formal specification languages, refinement, concurrency

Disciplines
Computer Sciences | Systems Architecture | Theory and Algorithms

Comments
Copyright © Gary T. Leavens, Tim Wahls, and Albert L. Baker, 1996. All rights reserved.

This article is available at Iowa State University Digital Repository: http://lib.dr.iastate.edu/cs_techreports/123
Formal Semantics for Structured Analysis Style Data Flow Diagram Specification Languages
Gary T. Leavens, Tim Wahls, and Albert L. Baker
TR #96-16
December 1996

Keywords: structured analysis, data flow diagram, operational semantics, formal specification, firing rule, store.


Submitted for publication.
Copyright © Gary T. Leavens, Tim Wahls, and Albert L. Baker, 1996. All rights reserved.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA
Formal Semantics for Structured Analysis Style Data Flow Diagram Specification Languages

Gary T. Leavens*
Department of Computer Science, 229 Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1040 USA
leavens@cs.iastate.edu

Tim Wahls
Department of Mathematical and Computer Sciences
E258 Olmsted, Penn State Harrisburg,
777 W. Harrisburg Pike, Middletown, PA 17057 USA
wahls@psu.edu

Albert L. Baker†
Department of Computer Science, 228 Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1040 USA
baker@cs.iastate.edu

December 3, 1996

Abstract

Using operational semantic techniques, we present a formal semantics for an extended variant of structured analysis style data flow diagrams. This semantics is intended to serve as a semantic foundation for many different specification languages that specify concurrent systems using a graphical notation similar to data flow diagrams. Besides allowing one to specify how information is processed, it allows one to specify the dynamic behavior of a concurrent system. We discuss various semantic issues, including the need for a two-step firing rule and how the semantics supports the notion of refinement.

1 Introduction

An approach to the development of software systems which has enjoyed wide-spread use in the software engineering community is Structured Analysis (SA) [1] [2] [3] [4]. Within SA one specifies a data model using an Entity-Relationship Diagram (ERD) and a data dictionary, and the process by a Data Flow Diagram (DFD) [5]. Because DFDs are widely-used [6], many tools support their development. There are at least three attributes of DFDs that are appealing to software engineers:

- they have a graphical representation,
they are hierarchical, thereby supporting the kind of modular decomposition that programmers view as essential, and

- they are informal.

Since DFDs do not have a precise semantics, a DFD (even when combined with a ERD) cannot serve as a formal specification of the functionality of a software system. In addition, DFDs are not even intended to capture the dynamic behavior of a software process.

Various researchers have proposed ways to use SA techniques either as a first step towards extracting a formal specification, or by augmenting SA techniques with the goal of making them more precise [7]. For example, Fraser, Kumar, and Vaishnavi [8] and Larsen, et al. [9] extract a VDM [10] [11] specification from a DFD and a data dictionary. As another example, Semmens and Allen [12] extract a Z [13] [14] [15] specification from an application’s ERD and DFD. See [16] for a survey that includes other such examples.

For this paper, the more relevant techniques are those that use formal notations to supplement SA techniques. For example, Wing and Zaremski [17] augment SA specifications (especially the data dictionary), with specifications in the Larch Shared Language [18]. Several researchers have used VDM specifications to augment the data dictionary and to specify the behavior of bubbles [9] [19] [20] [21]. As another example, France [5] [22], has specified the types in a data dictionary using Z notation, and has used Z to augment the specifications of data stores and global state invariants. These extensions are relevant, because they indicate that there is a desire for integration of DFDs and formal methods.

These supplements to SA techniques motivate the problem we address in this paper: what would be a suitable formal model that extends the standard concepts of DFDs to allow the description of a concurrent system’s dynamic behavior? Our solution to this problem, we claim, provides a semantic foundation for:

- defining extended DFD specification languages that would allow for formal specification of both data transformations and dynamic behavior, and the consequent ability to do formal reasoning and validation,

- comparing the semantics of various extended DFD specification languages, and

- using DFDs informally in ways that are more expressive and precise.

An additional benefit of our efforts is that, by focusing on the semantics of an extended variant of DFDs, we explore much of the space of possible specification languages that use DFD notations and specify dynamic behavior; such an exploration would not be possible if we were presenting a particular specification language.

Some may object to this effort on the grounds that its intention to support extended DFD specification languages is misguided. They would say that a large part of the appeal of DFDs is their informality, which allows them to be used during early stages of requirements analysis and specification. But having a formal DFD specification language will certainly not prevent anyone from drawing pictures on paper; what it will do is allow the possibility of taking such paper sketches and formalizing them without a complete change of notation. We also believe that an understanding of our semantics will aid both the design of extended DFD specification languages and aid the work of systems analysts who wish to be more precise in their use of DFDs.

Due to space considerations, we will not give the details of an extended DFD specification language (some preliminary ideas are, however, found in [23]), or a translation from such
a language into our semantic model. What we offer instead is the definition of the target of such a translation (the semantic model itself). To illustrate the semantic model, we also offer an example of both ends of this translation (the input and the resulting model). The reader can use this example to judge the suitability of our model. However, the focus of this paper is the formal model, a variant of DFDs, and its formal semantics.

Because we are giving a semantic foundation to extended DFD specification languages, we do not limit ourselves to exactly the traditional notations and concepts used in DFDs. (Such notations and concepts are fraught with ambiguity in any case.) Instead, we follow the example of workers in semantics of programming languages, who extract from real programming languages a set of core concepts, into which a real programming language could be translated. No one expects a theoretical core language (like the λ-calculus) to be of practical value in real programming. Instead, a standard way to judge such theoretical core languages is to see if they can capture (by translation) the meaning of all the constructs of relevant real languages. Thus, our intention is to define a model that can act as a theoretical core for DFD-like specification languages. The model should be judged by whether it can serve as a translation target for traditional DFDs and extended DFD-like specification languages. It should not be judged by how exactly it matches the traditional DFD notation.

The main extension to the traditional notions of DFD specification is the specification of dynamic behavior; that is, our model gives a formal description of a DFD's firing rules. We believe that with such an extension, DFDs may be useful for the specification of concurrent and distributed systems. In such applications, the precise dynamic behavior of the system is more important than when a DFD is used to help design a single-processor program.

In overview, the meaning an extended DFD specification would be assigned in our model is a set of sequences of configurations of the DFD. A configuration of a DFD tells the state of each process and flow. A sequence of configurations represents a possible execution of the DFD. The model uses sets of such sequences to handle concurrency and nondeterminism that may be allowed by the specification.

In addition to the modeling of dynamic behavior, we believe that the following aspects of our semantics are interesting in the sense of finding a smaller theoretical core for DFDs.

1. DFD terminators (external entities) have a specified behavior, which is unusual (although found in [5]); however, if no constraints on the terminators are desired, then the specification can simply permit arbitrary behaviors.

2. DFD stores, which are often seen as abstractions for files, can be modeled using only data flows that hold a single (possibly compound) value. This also provides a model for shared variables in a straightforward manner. (One could also use flows that hold queues of values as a model of message passing in distributed systems.)

3. Non-primitive bubbles in a hierarchically decomposed DFD are modeled by allowing such a bubble to fire concurrently with itself. That is, the model allows a bubble to read again before it has written its output. This allows a single bubble to act like a system of interacting bubbles, and thus the dynamic behavior of a single bubble can be refined by a system of bubbles in a hierarchical DFD.

The rest of the paper describes our semantics for our theoretical core DFDs. (Thus, when we say “DFD” below we usually mean our core theoretical variant of the DFD notion, not the traditional notion.) This paper is organized as follows. Section 2 below describes
the rationale for the important design decisions taken in formulating the semantic model. Section 3 describes the structure of DFDs. Section 4 describes configurations of DFDs which are the basis of the operational semantics. Section 5 describes the meaning of a P-spec. Section 6 presents the basic operational semantics. Following that, Section 7 presents some discussion, including a discussion of related work.

2 Semantic Modeling Considerations

In this section we treat several issues in the design of a formal model for the dynamic behavior of DFDs. The main design criteria is that the model should be general enough to serve as the translation target of several different extended DFD specification languages.

2.1 What Types of Values on Flows?

In data processing, traditional programming languages support either a fixed set of data types, such as integers, characters, and compound types such as strings, records, and arrays. However, there seems to be no reason for a semantic model to limit the types of values that can appear on flows to such a fixed set. In our model, the types of values can be arbitrary abstract data types (ADTs), including stacks, B-trees, etc., of arbitrary complexity.

2.2 What Kinds of Flows to Model?

In SA, DeMarco [1] and others (e.g., [24]) make a distinction between “discrete” and “continuous” flows. Some authors seem to consider continuous flows as continuous in the sense of calculus (i.e., real-valued, differentiable). DeMarco and Ward seem to think of them as continuously existing (i.e., like a shared variable) even when read.

Since the formal model allows a DFD to process and transmit values regardless of their type, there seems to be no need to make special provision for datatypes such as the real numbers. On the other hand, the difference between a shared variable and a queue is fundamental, and needs to be directly supported by the formal model. Hence our formal model supports this distinction. However, to avoid confusion, we use different terminology [25]. Flows labeled with a single arrowhead are called consumable flows; these can be considered to be unbounded FIFO queues. Flows labeled with a double arrowhead are called persistent flows; these can be considered to be shared variables.

For the same reason, the formal model does not make special provision for “control flows”. This is because one can model a DFD that has a control flow by a consumable flow of type Signal, where Signal is a type that has only one element. (One could also use a persistent flow and a two-element type to indicate that the signal is “on” or “off”.)

2.3 How Should Stores be Modeled?

In traditional SA, a store is a passive holder of data [1] [25], much like a file in data processing. Multiple bubbles can access a single store for both reading and writing, but the store itself does not transform data.

Since persistent flows in our model act like shared variables, following a suggestion by Coleman [25, Figure 8.2] we keep the set of modeling concepts small by modeling stores as persistent flows. Since multiple bubbles in a DFD can have read or write access to a single
store, this decision requires that the formal model allow a persistent flow to have multiple source and target bubbles.

Instead of our chosen semantics, we could have made stores a primitive in the semantics, and translated persistent flows in a specification into stores in the model. We chose to say that stores are translated into persistent flows mainly to emphasize the idea of this equivalence. Making persistent flows primary in the model also seems to make for a more uniform semantic model, as flows only go between bubbles, not between stores and bubbles. (Making a store a kind of bubble is another option, but since stores are not active, that model seems to be misleading.)

In a practical DFD specification language, one should certainly include the traditional store notation, as we do in our examples. In such a language, one might also want to include persistent flows, even though the semantics is equivalent, because stores and persistent flows may have different connotations.

Stores often represent a collection of data in traditional DFDs. When desired, this is easily modeled by a persistent flow of type set or sequence. Similarly, stores containing values of an ADT may be modeled by a flow whose type is that ADT. In other words, the type system used in an extended DFD specification language, and not that used in our semantics, is the only thing that would impose a limit on the kind of stores that can be specified.

2.4 What Should the Initial Value be for a Flow?

Some authors (e.g., [19] [20]) use a formal model in which each flow in a DFD starts off empty (i.e., is uninitialized). However, if one considers a specification language where users can specify initial values for all flows, such a default makes it more difficult to translate that specification language into the formal model. One could try to have each bubbles initialize its outflows upon noting that its inflows are empty, but that fails for sources (as they have no inflows), and also fails when all of a bubble’s inflows are initialized by the initialization actions of other bubbles.

We require, therefore, that the initial values of flows be given as part of the specification of a DFD in our model. (If a specification language has a default for initial values of flows, then that is easily accommodated in its translation.)

2.5 Should a Bubble Fire Atomically?

In our formal model, we use a two-step firing rule for bubbles, instead of considering the firing of a bubble to be atomic. That is, when a bubble “fires”, it does so in two steps: first it reads (some of) its inputs, and then it computes and writes (some of) its outputs. This two-step firing rule allows the modeling of concurrent firing of bubbles and time delays. (However, we do not model real-time aspects of time delays, as in [3] [19] [20].)

To see why a two-step firing rule is necessary, suppose that bubbles were able to read their inputs and compute and write their outputs in one atomic step. We will show that we would be unable to model time delays and race conditions that can occur in systems where processes take a finite amount of time to compute results.

For example, consider the DFD shown in Figure 1. Suppose that the bubble Test periodically changes the value on the persistent flows named y and z by setting both to the same value in the sequence 0, 1, 2, 3, .... Suppose that the bubble Copy copies its input on z to x, and that the initial values of x, y, and z are 0. Finally, observe the results through
Figure 1: Data flow diagram showing a race condition.

the bubble *Observe*; that is observe a sequence of pairs of the form \((x = x_i, y = y_i)\). Under an atomic firing rule, if we observe a pair \((x = x_i, y = y_i)\), then the next pair we observe, \((x = x_{i+1}, y = y_{i+1})\), must satisfy one of the following conditions:

- \(x_{i+1} = x_i\).
  This will be the case if *Copy* has *not* fired, since only *Copy* writes the flow named *x*.

- \(x_{i+1} \geq y_i\).
  This will be the case if *Copy* fired. It is impossible for \(x_{i+1}\) to be less than \(y_i\) (the previous value on \(y\) and \(z\)), because the value on \(y\) and \(z\) only increases, and if *Copy* fires it atomically copies to \(x\) the value on \(y\) and \(z\), which must be at least \(y_i\).

The behavior of the DFD in Figure 1 is different if *Copy* takes some amount of time to fire, because then there will be a delay between when it reads its inputs and when it writes its outputs. During this delay, other bubbles may fire. To be concrete, consider the initial configuration (where \(x\), \(y\), and \(z\) are 0). In this configuration one can observe the pair \((x = 0, y = 0)\). Suppose that *Test* then changes \(y\) and \(z\) to 1. Now suppose *Copy* reads the value 1 from \(z\). Suppose that *Test* changes \(y\) and \(z\) to 2. In this configuration one can observe the pair \((x = 0, y = 2)\). Finally, suppose that *Copy* writes out the value it read; that is, it sets \(x\) to 1. In this configuration one can observe the pair \((x = 1, y = 2)\). The sequence of pairs observed in this execution would be impossible with the atomic firing rule, because the value of \(x, 1\), is neither the previous value of \(x, 0\), nor is it greater than or equal to the previous value of \(y\) (and \(z\)), 2. We emphasize that this sequence of observations would be possible in a world where bubbles take a finite time to execute and where executions can overlap. Hence the necessity of non-atomic firings in an adequate formal model.

2.6 How Can a Bubble be Refined?

With a two-step firing rule, one allows for bubbles to fire concurrently with other bubbles. However, in a hierarchical DFD, a single bubble may be refined into a sub-DFD consisting of several bubbles. If the bubbles in this sub-DFD can fire concurrently with each other, they can produce behaviors that seem impossible to achieve with the single original bubble.
Before giving an example, let us define a few terms. Following [26], a scenario of a DFD is a pair consisting of a sequence of values presented to the DFD as inputs (i.e., on the flows coming from the external inputs) and an associated output sequence of values (i.e., on the flows into the external outputs), augmented with a partial order on the values showing causality relationships. That is, values \( x \) and \( y \) are related under this ordering if \( x \) must appear before \( y \). We present some example scenarios shortly. A scenario set is the set of all possible scenarios associated with a particular DFD. A DFD \( d_2 \) refines a DFD \( d_1 \) if the scenario set of \( d_2 \) is a subset of the scenario set of \( d_1 \). That is \( d_2 \) refines \( d_1 \) if every scenario of \( d_2 \) is a possible scenario of \( d_1 \). A refinement may be more deterministic than the DFD it refines.

The problem considered in this subsection, then, is how to formally model DFDs so that the behavior of lower levels of a hierarchical DFD is a refinement of the behavior of higher-level DFD. This is important so that the higher levels of a hierarchical model can serve as abstractions of the behavior of lower levels.

To see the potential problems, consider the refinement shown in Figure 2. To get to the point faster, we describe what the bubbles in this figure do informally. In the top part of Figure 2, the bubble \( H \) is a top-level specification of the function

\[
\text{from} = \text{if } \text{to} \leq 2 \text{ then } \text{to}^2 \text{ else } \text{to}^3
\]

where \( t \text{o} \) is the value read from the flow named \( \text{To} \) and \( \text{to}^2 \) or \( \text{to}^3 \), as appropriate, is written to the flow \( \text{From} \). A scenario associated with this first DFD and the inputs 2 followed by 3 (i.e., \( \text{Term1} \) produces first 2, then 3), is given in Figure 3. The arrows in the figure represent the causality relationship — that is, that the value at the source of the arrow must appear
before the value at the target can. The values 2 and 3 appear on the To and the values 4 and 27 on the flow From. Because Bubble H must read and process the value 2 before it can read the value 3, this is the only scenario resulting from this input sequence.

The bottom part of Figure 2 is intended as a refinement of the top part, where the bubble H is expanded into a sub-DFD. Bubble Split sends inputs less than or equal to 2 to bubble Square, and other inputs to bubble Cube. Square and Cube each read their inflow, compute the appropriate value, and write it to their outflow. Bubble Merge reads from either of its inflows, and places the value read on the flow From.

Intuitively, the refinement should behave exactly like the original DFD. However, if the flow To contains integers 2 and then 3, then two scenarios result, as shown in Figure 4. The first scenario is the same as that of Figure 3, while the second results from the case where Merge reads the value 27 from FromCube and writes it to From before reading 4 from FromSquare. While this is clearly a contrived example, similar situations can arise any time one bubble is refined by multiple bubbles.1

If the bubble H could somehow hold the first value it consumes while consuming and producing output with the second, then both scenarios of the refinement would also be possible scenarios of the original DFD. Thus, in our formal model of DFDs, each bubble is allowed to fire concurrently with itself. That is, a bubble can consume from its input flows multiple times without producing any output, and when the bubble does produce output, the output can be produced from the input provided by any of the previously consumed inputs, not just the first.

There is one complication, however, in choosing a semantic model that allows a bubble to fire concurrently with itself. Suppose one wants to specify a bubble, at the lowest level

1In this simple example, the partial order on values in a scenario may appear to be unnecessary. However, Brock and Ackerman [26] have shown that two “network computations” with the same history relation (scenario set minus the partial order) can not always be substituted for each other in a larger network computation without changing the history relation of that larger computation. Their proof translates easily to DFDs.
of a hierarchy, that is not supposed to fire concurrently with itself. That is, how can one specify a sequential bubble?

One way to model sequential bubbles would be to have two kinds of bubbles in the model: atomic bubbles (which cannot fire concurrently with themselves) and nonatomic bubbles. However, this seems somewhat ad hoc, as it would not allow the possibility of specifying a DFD with some other execution pattern (such as reading twice, then writing twice).

The more general solution to this problem, which we have chosen for our model, is to allow one to specify more precisely when a bubble is allowed to read and when it is allowed to produce output. The formal model allows one to specify whether a bubble is enabled for reading based on two pieces of information: the state of its input flows and what it may have already read. The formal model allows one to specify when a bubble is ready to produce output based on what it may have previously read, and in addition allows the specification to select from the previously read inputs a set of inputs that are eligible to be used to produce output. A sequential bubble is modeled by having the bubble only be enabled for reading when it has no previously read inputs that are still available for processing, and by having the bubble select its only available input whenever there is such a single input available.

2.7 Are Firing Decisions Local?

One can imagine specifying the behavior of a DFD by considering global state information (i.e., that two particular bubbles cannot fire concurrently, or that a bubble can only fire if there is a certain value in some store that it cannot read). The alternative is to only permit a specification to take information that is local to a bubble into account.

While it is clearly more expressive to permit global information to be considered, we believe that it is more in the spirit of the DFD concept to only allow local information to be used. The reason is that this forces synchronization information to be recorded in the structure of a DFD as a data flow, making the extended DFD specification clearer to those reading such a specification.

Having dealt with the various design issues behind our formal semantic model of extended DFDs, we now turn to the details of the model itself.

3 Structure of a Data Flow Diagram

For the purposes of this paper, the structure of a DFD is modeled as a pair of sets, $(B, F)$. The set $B$ represents the bubbles in the pictorial form of a DFD. The set $F$ representing the labeled flows of a DFD. Jumping ahead a bit, each bubble will denote a process, and each labeled arrow denotes a potential information movement. The formal model of a DFD's structure is summarized in Table 1. An explanation of the formal model and how it correspondence with the usual DFD pictures is given below.

The graphical syntax for DFDs also has stores. In our examples, stores are drawn as parallel lines enclosing a label. However, these stores are formally modeled as persistent flows, and hence there is no part of our formal model explicitly devoted to modeling stores.

Although "data dictionaries" that provide additional information about a DFD are not strictly part of its structure, we also use this section to describe how data dictionaries fit into our formal model.


<table>
<thead>
<tr>
<th>member ∈ Name</th>
<th>= description</th>
</tr>
</thead>
<tbody>
<tr>
<td>b ∈ B</td>
<td>= a set (of bubble names)</td>
</tr>
<tr>
<td>bs ∈ PB</td>
<td>= PowerSet(B)</td>
</tr>
<tr>
<td>fn ∈ FLOWNAME</td>
<td>= a set (of flow names)</td>
</tr>
<tr>
<td>T ∈ TYPES</td>
<td>= a set (of type names)</td>
</tr>
<tr>
<td>p ∈ P</td>
<td>= {persistent, consumable}</td>
</tr>
<tr>
<td>f ∈ F</td>
<td>= PB × FLOWNAME × TYPES × PB × P</td>
</tr>
</tbody>
</table>

Table 1: Domains describing the structure of a DFD.

3.1 An Example DFD Picture and its Formal Model

An example DFD for an accounts receivable system is pictured in Figure 5. Informally, the idea behind this simplistic DFD is that when a customer communicates a need to a member of the sales staff, a master list of customer information is updated, and either a credit order or a cash order is generated. Both kinds of orders generate changes in the accounts receivable database, but credit orders also result in a billing. In the process of billing, a clerk sends a bill to the customer through the mail, and a copy of the bill is sent to update the accounts receivable database. A customer may also send a payment, consisting of money and the bills that are being paid. The credit from the payment to be applied to each account is used to update the accounts receivable database; that is, the collections process sorts out how much of the payment is to be applied to each account.

3.1.1 Bubbles and Terminators

A named bubble in the graphical notation is represented by its bubble name in the formal model. So for the DFD in Figure 5 the set B contains just the following names: Customers, Generate-Sales-Order, Bill-Customer, Keep-Customer-Accounts, Collections, and Mail.

The terminators (external entities) of a DFD are not treated differently by our formal model. In Figure 5, the terminators are drawn using rectangles instead of ovals; that is, the bubbles Customers and Mail are its terminators. (In that example, the terminators are also the sources and sinks of the DFD considered as a directed graph, but that is not required by our model.) Considering terminators to be part of the DFD may strike some as unusual, but it will be seen that they can be treated as bubbles, although in the usual case no interesting constraints will be specified on their behavior.

3.1.2 Flows and Stores

In our example drawings, all flows, except those to and from stores, are labeled with three pieces of information: either a single or double arrowhead, a flow name fn ∈ FLOWNAME, and a type name T ∈ TYPES. Neither the flow name nor the type alone are sufficient to uniquely identify a flow in our example; having both pieces of information eliminates a common source of ambiguity [25].\(^2\)

\(^2\)Some extended DFD specification languages may permit flows with the same name and type to appear in multiple places in a DFD; this does not cause a problem for the formal model of flows, in which the source and target bubbles also help identify it.
Figure 5: An example of one graphical notation for an extended DFD. This DFD is part of the specification of an accounts receivable system.
In our examples, we draw persistent flows (shared variables) with double arrowheads, and consumable flows (FIFO queues) with single arrowheads. For example, the flow what of type Need is a consumable flow.

Each store and all of the flows into it and out from it are formally modeled as a single persistent flow. The name and type of the store are taken as the name and type of formal model’s persistent flow. In the example drawing, there are three stores, each with a double arrow into and out from it. Thus there are three persistent flows in the formal model, one named custs, one named log, and one named c-a-data.

In general there can be multiple flows to and from a single store. To accommodate this our formal model allows a flow to have multiple sources and targets.

Thus a flow is formally modeled by a 5-tuple, \((bs_1, fn, T, bs_2, p)\), which represents a flow named \(fn\) going from the bubbles in the set \(bs_1\) (the sources) to the bubbles in the set \(bs_2\) (the targets), carrying information of type \(T\); this flow is persistent if \(p = \text{persistent}\) and consumable if \(p = \text{consumable}\). (We will only consider flows that have at least one source and target.) For example, the flow from Collections to itself is formally described as the 5-tuple \((\{\text{Collections}\}, \text{log}, \text{Money-and-Acct-List}, \{\text{Collections}\}, \text{persistent})\). Since such 5-tuples tend to be rather unwieldy, for all the flows in Figure 5 we give abbreviations in Table 2. Abbreviations are based on the flow name, and the two flows named credit are distinguished by appending an abbreviation for their targets.

In a persistent flow, the shared variable that the flow represents can be written by each bubble at the source of the flow, and read by each target bubble [25]. Reading from a persistent flow does not change the information in the flow. In the formal model of our example, the only persistent flows go from bubbles to themselves, because they are used to model local stores. For example, \(f_{\text{custs}}\) is used to model the store attached to the bubble Generate-Sales-Order.

In a consumable flow, the source bubble enters tokens of information at the tail of the queue that the flow represents, and the target bubble removes information tokens from the head of the queue. (The dynamic behavior of flows is discussed in more detail below.)
Table 3: Auxiliary Functions for describing the structure of a DFD, where Set is the class of all recursive sets.

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sources</td>
<td>F -&gt; PowerSet(B)</td>
</tr>
<tr>
<td>FlowName</td>
<td>F -&gt; FLOWNAMES</td>
</tr>
<tr>
<td>TypeOf</td>
<td>F -&gt; TYPES</td>
</tr>
<tr>
<td>Targets</td>
<td>F -&gt; PowerSet(B)</td>
</tr>
<tr>
<td>Consumable</td>
<td>F -&gt; Boolean</td>
</tr>
<tr>
<td>Inputs</td>
<td>B -&gt; PowerSet(F)</td>
</tr>
<tr>
<td>Outputs</td>
<td>B -&gt; PowerSet(F)</td>
</tr>
<tr>
<td>TypeMeaning</td>
<td>TYPES -&gt; Set</td>
</tr>
</tbody>
</table>

3.2 Auxiliary Functions for Flow Components

It will be convenient to have some auxiliary functions for accessing the components of a flow. These functions are summarized in Table 3.

If \( f = (b_1, fn, T, b_2, p) \), then Sources\( (f) = b_1 \), FlowName\( (f) = fn \), TypeOf\( (f) = T \), and Targets\( (f) = b_2 \). The value of Consumable\( (f) \) is true if \( p = \text{consumable} \) and is false otherwise. For example, if \( f_{\text{payment}} \) is the flow from Figure 5 described in Table 2, then Sources\( (f_{\text{payment}}) = \{\text{Customers}\} \), FlowName\( (f_{\text{payment}}) = \text{payment} \), TypeOf\( (f_{\text{payment}}) = \text{Money-and-Bills} \), Targets\( (f_{\text{payment}}) = \{\text{Collections}\} \), and Consumable\( (f_{\text{payment}}) = \text{true} \).

A flow \( f \) is said to be an input flow of a bubble \( b \) if \( b \in \text{Targets}(f) \). We write Inputs\( (b) \) for the set of all of \( b \)'s input flows. Similarly, \( f \) is an output flow of \( b \) if \( b \in \text{Sources}(f) \). We write Outputs\( (b) \) for the set of all of \( b \)'s output flows. In our example, Inputs\( (\text{Mail}) = \{f_{\text{invoice}}\} \) and Outputs\( (\text{Mail}) = \{\} \).

3.3 Data Dictionaries

A “data dictionary” is often associated with a DFD, and sometimes describes all the flows. An example is found in Figure 6. However, for the formal model, all that is needed from the data dictionary is that it define the meaning of each type used in a DFD. Since very little of the semantics of types is needed for this, in the formal model a type is considered to be just a set of objects. We denote the set of objects associated with a type \( T \) by TypeMeaning\( (T) \). For example, Figure 7 might be the type meaning function extracted from the data dictionary in Figure 6.\(^3\)

In Table 3, Set denotes the class of all recursive sets. This suffices for normal use of DFDs, but more exotic domains could be used to model types such as higher-order functions or lazy streams.

4 Configurations of Data Flow Diagrams

In several places in the semantics, we shall be manipulating (possibly undefined) sequences of objects, that is elements of \((\text{OBJECTS}^*)_\perp\).\(^4\) For example, in the semantics of flows below,

\(^3\)The notation \( \text{Char}^* \) means the set of all finite sequences of characters. The notation \( \text{PowerSet(\text{Char}^*)} \) means the set of all sets of such sequences. We ask readers familiar with such notational conventions to bear with us as we try to make this paper accessible to a wider audience.

\(^4\)We write \( X_\perp \) for the set \( X \cup \{\perp\} \). The notation \( \perp \) means “no information”.

13
Data Dictionary:

type Money = real;
type Account = string;
type Need = tuple of (thing: string, cust: string);
type Bill = tuple of (name: string, goods: string,
                        acct: Account, amount: Money);
type Money-and-Bills = tuple of (amt: Money, bill: Bill);
type Money-and-Acct = tuple of (amt: Money, acct: Account);
type Money-and-Acct-List = sequence of Money-and-Acct

type Order = tuple of (goods: string, amt: Money);
type Cust-DB = set of string;
type Acct-DB = set of tuple of (name: string, acct: Account,
                                balance: Money);

Figure 6: An example of one notation for a data dictionary.

\[
\begin{align*}
\text{TypeMeaning}_e(\text{Money}) & = \text{Real} \\
\text{TypeMeaning}_e(\text{Account}) & = \text{Char}^* \\
\text{TypeMeaning}_e(\text{Need}) & = \text{Char}^* \times \text{Char}^* \\
\text{TypeMeaning}_e(\text{Bill}) & = \text{Char}^* \times \text{Char}^* \times \text{TypeMeaning}_e(\text{Account}) \\
& \quad \times \text{TypeMeaning}_e(\text{Money}) \\
\text{TypeMeaning}_e(\text{Money-and-Bills}) & = \text{TypeMeaning}_e(\text{Money}) \times \text{TypeMeaning}_e(\text{Bill}) \\
\text{TypeMeaning}_e(\text{Money-and-Acct}) & = \text{TypeMeaning}_e(\text{Money}) \\
& \quad \times \text{TypeMeaning}_e(\text{Account}) \\
\text{TypeMeaning}_e(\text{Money-and-Acct-List}) & = (\text{TypeMeaning}_e(\text{Money-and-Acct}))^* \\
\text{TypeMeaning}_e(\text{Order}) & = \text{Char}^* \times \text{TypeMeaning}_e(\text{Money}) \\
\text{TypeMeaning}_e(\text{Cust-DB}) & = \text{PowerSet}(\text{Char}^*) \\
\text{TypeMeaning}_e(\text{Acct-DB}) & = \text{PowerSet}\left(\text{Char}^* \times \text{TypeMeaning}_e(\text{Account}) \times \text{TypeMeaning}_e(\text{Money})\right)
\end{align*}
\]

Figure 7: Example of a \text{TypeMeaning} function, \text{TypeMeaning}_e.
we treat sequences of objects as FIFO queues. To do this it is convenient to define some constants and operations. The types of these are summarized in Table 4. They are defined to satisfy the equations given in Figure 8, for all $q \in OBJECTS^*$ and $o \in OBJECTS$.

In dealing with $\bot$ in example formulas, we assume that all boolean functions are strict; for example, $(\neg \bot) = \bot$ and $(true \lor \bot) = \bot$.

Our model of the execution of a DFD consists of bubble computations and information moving from one bubble to another along the flows. A configuration is a snapshot of such an execution, taken at certain well-defined points; it is similar to a marking of a Petri net [27].

A configuration records the information that bubble has read from its input flows (if any), and the information present on each flow. The formal model of configurations, the set $\delta$, is summarized in Table 5, and will be explained below.

Recall from Section 2.5, that to allow non-atomic firings, bubbles do not read from their inputs and write their outputs in one step. Instead, firing consists of two steps: reading from inputs, and then computation and writing of outputs. To allow bubbles to fire concurrently with themselves, when a bubble reads from its inputs it records what it read in a sequence (i.e., in a list). A sequence (or some such data structure) is needed because a bubble's inputs may present the same values to the bubble multiple times, and a set would not store such repeated values. When a bubble later produces output, it selects one record of its inputs and uses that to produce its outputs. The selection does not necessarily have to be made in FIFO order, but using a sequence facilitates the specification of bubbles that process their inputs in FIFO order.
Figure 8: Specification of the operations on sequences. These hold for all \( q \in OBJECTS^* \) and \( o \in OBJECTS \).
\[
\begin{align*}
    r_e(\text{Customers}) & = \langle \rangle \\
    r_e(\text{Generate-Sales-Order}) & = \langle \rangle \\
    r_e(\text{Bill-Customer}) & = \langle \lambda f . \text{if } f = f_{\text{creditBC}} \text{ then } \text{co} \text{ else } \perp \rangle \\
    r_e(\text{Keep-Customer-Accounts}) & = \langle \rangle \\
    r_e(\text{Collections}) & = \langle \lambda f . \text{if } f = f_{\text{payment}} \text{ then } \text{pmt} \text{ else if } f = f_{\log} \text{ then } \langle \rangle \text{ else } \perp \rangle \\
    r_e(\text{Mail}) & = \langle \rangle \\
\end{align*}
\]

Figure 9: Example of a Read function, \(r_e\), where \(\text{co} \in \text{TypeMeaning}_{(\text{Order})}\), \(\text{pmt} \in \text{TypeMeaning}_{(\text{Money-and-Bills})}\), and \(\langle \rangle \in \text{TypeMeaning}_{(\text{Money-and-Acct-List})}\).

4.1 Information that a Bubble has Read from its Inputs

The record of what a particular bubble has read from its inputs is recorded as a sequence of functions of type \(\text{WhatRead} = F \rightarrow \text{OBJECTS}_\perp\); that is, functions from flows to either a value or \(\perp\).

In a particular configuration, there is a sequence of such functions for each bubble in the DFD. The formal model of what bubbles have read in a configuration is thus a curried function from bubble names to sequences of type \(\text{WhatRead}\). We use \(\text{Read}\) to stand for the set of all such functions.

In a valid configuration, not just any function of type \(\text{Read}\) is acceptable. The typing information on flows must be obeyed, and also the information recorded for a bubble must be local to that bubble.

The typing restriction is as follows. Suppose that \(r \in \text{Read}\) is part of a valid configuration, \(b\) is a bubble name, and \(f\) is a flow. Then \(r(b)\) is of type \(\text{WhatRead}\*\) — a sequence of functions from flows to information, representing what \(b\) has read from its input flows but not yet used to produce outputs. Each \(s\) such that \(s\) is in the sequence \(r(b)\) must be such that \(s(f)\) is either \(\perp\) or an element of the type of the flow \(f\):

\[
\forall s . \text{IsIn}(s, r(b)) \Rightarrow s(f) \in \text{TypeMeaning}(\text{TypeOf}(f))_\perp.
\]

The locality restriction is as follows. Suppose that \(r \in \text{Read}\) is part of a valid configuration; then for each bubble \(b\), each \(s\) that is a member of the sequence \(r(b)\) should only be defined on flows that are inputs to \(b\). This is stated formally as follows. For each \(b \in B\), \(f \in F\), and \(s \in \text{WhatRead}\):

\[
(\text{IsIn}(s, r(b)) \land f \not\in \text{Inputs}(b)) \Rightarrow (s(f) = \perp).
\]

An example of a \(\text{Read}\) function is given in Figure 9.\(^5\)

The function \(r_e\) describes part of a configuration where the bubble \text{Bill-Customer} has read the order \(\text{co}\) from the flow \(f_{\text{creditBC}}\) (see Table 2), and where \text{Collections} has read the payment \(\text{pmt}\) from \(f_{\text{payment}}\) and an empty list from \(f_{\log}\).

4.2 Information on a Flow

The information on the flows of a DFD is captured by a function from flows to a finite sequence of objects of the appropriate type. A persistent flow must be mapped to a se-

\(^5\)The notation \(\lambda f . \epsilon\) denotes a function that takes an argument, and returns the value of the expression \(\epsilon\) with the argument substituted for \(f\) [28] [29].
\[
fs_e(f) = \begin{cases} 
\{ cdh \} & \text{if } f = f_{\text{custs}} \\
\{ arlh \} & \text{if } f = f_{\text{c-a-data}} \\
\emptyset & \text{if } f = f_{\text{log}} \\
\{ \} & \text{otherwise,}
\end{cases}
\]

Figure 10: An example \textit{FlowState} function, \(fs_e\), where \(cdh \in \text{TypeMeaning}_e(\text{Cust-DB})\), \(arlh \in \text{TypeMeaning}_e(\text{Acct-DB})\), and \(\{ \} \in \text{TypeMeaning}_e(\text{Money-and-Acct-List})\).

sequence of length zero or one; when the sequence is empty, it represents an uninitialized persistent flow. (An alternative would be to use \(\bot\) for uninitialized persistent flows.) We use \textit{FlowState} to stand for the set of all such functions \(fs\). In a valid configuration, the \textit{FlowState} component, \(fs\), must be such that the following conditions are satisfied for all \(f \in F\):

- \(\text{TypeOf}(fs(f)) = \text{TypeOf}(f)^*\), and
- \(\neg \text{Consumable}(f) \Rightarrow \text{Length}(fs(f)) \leq 1\).

An example \textit{FlowState} function is given in Figure 10. The function \(fs_e\) describes part of a configuration where there are no values on any of the consumable flows, and various values \((cdh, arlh, \text{and } \{\})\) on the persistent flows that model the stores of Figure 5.

4.3 Configurations

A configuration summarizes all the information about the state of a DFD described above.

Formally, a \textit{configuration} of a DFD is a pair, \((fs, r)\), where \(fs \in \text{FlowState}\) and \(r \in \text{Read}\). The set of all configurations is denoted by \(\mathcal{C}\).

An example configuration of the formal model of the DFD in Figure 5 is the pair \((fs_e, r_e)\), where \(fs_e\) and \(r_e\) are as described in Figures 10 and 9.

5 Meaning of a P-Spec

The things that the bubbles in a DFD can do are specified by a \textit{P-spec} (short for “process specification”). For example, part of an example P-spec is given in Figure 11. In this notation, the bubble Collections is not allowed to fire concurrently with itself. (Let us suppose that the keyword \text{system} would be used instead of \text{process} to permit the bubble to fire concurrently with itself.)

The form given in Figure 11 is just one possible syntax for a P-spec. Some authors use finite state machines, and others use a formal specification language such as VDM or Z. Because our model is intended to be a target for any such specification language, it must be more abstract and general. Note that we cannot give a general translation of a P-spec into our model, as this depends on the particular form and semantics chosen for the presentation of P-specs.

To achieve the necessary abstraction and generality, our formal model of a P-spec consists of four curried functions and an initial \textit{FlowState} map:

\[(\text{Enabled}, \text{Consume}, \text{Selected}, \text{Produce}, f_{\text{initial}})\].
process Collections:
  enabled when *payment
  ensures log′ = Enq(log, [amt: amt(payment), acct: acct(bill(payment))])
  and credit′ = [amt: amt(payment), acct: acct(bill(payment))]

Figure 11: Part of an example P-Spec that specifies the behavior of the Collections bubble. The bubble’s enabling condition is on the second line, which says it is enabled when there is input on the flow named payment. The third and forth lines say what outputs are produced by the bubble on the flows log (a store) and credit.

The four curried functions tell when each bubble in the DFD is enabled, what it consumes from its inputs when it is enabled, what previously read inputs it is willing to select for producing output, and what output it produces from a previously read input. We allow bubbles to be nondeterministic in what they consume and produce, as this is sometimes convenient in specifications.

We explain each part of the formal model of a P-spec below.

5.1 When are Bubbles Enabled for Reading?

In a given configuration, a bubble may or may not be enabled for reading. A bubble’s being enabled depends on the configuration’s flow state, because if the bubble needs to read consumable inputs, there must be a non-empty sequence of inputs available on the needed flows.

In our formal model, we allow enablement to depend on both the presence of values on input flows as well as on the values on such flows. Some specification languages might not allow enablement to depend on the values on flows, but the model is general enough to support those that do.

Enablement for reading also depends on what the bubble may have previously read, but has not yet used for output. For example, a sequential bubble would not be enabled after it has read once and before it has used the inputs it read to produce output.

In our formal model, the part of P-spec that tells when each bubble is enabled is captured by a mapping:

\[
\text{Enabled} : B \rightarrow ((\text{FlowState} \times \text{WhatRead}^*) \rightarrow \text{Boolean}_\bot) .
\]

This function is curried, so that for a bubble \( b \), \( \text{Enabled}(b) \) tells the pairs of flow states and sequences of \( \text{WhatRead} \) functions for which \( b \) is enabled. In a given flow state, \( fs \), there may be no information on a given flow; this is why \( \text{Enabled}(b)(fs, sq) \) is allowed to be \( \bot \); a bubble \( b \) is only considered enabled if \( \text{Enabled}(b)(fs, sq) = \text{true} \).

A valid \( \text{Enabled}(b) \) mapping should only depend on the states of the flows in \( \text{Inputs}(b) \), since those are the only flow states that it will be passed by the operational semantics. Formally, this locality condition is stated as follows. For all \( b \in B \), \( fs \in \text{FlowState} \), \( fs' \in \text{FlowState} \), and \( sq \in \text{WhatRead}^* \):

\[
(\forall f \in \text{Inputs}(b) : fs(f) = fs'(f)) \Rightarrow \text{Enabled}(b)(fs, sq) = \text{Enabled}(b)(fs', sq) .
\]
An example of an Enabled function for the DFD of Figure 5 is given in Figure 12. This says, for example, that Keep-Customer-Accounts is enabled if it has nothing that it has read but not processed, and if there is something on its persistent flow (the model of the store c-a-data), and if there is something on at least one of the three consumable flows into that bubble. Similarly, Collections is enabled if it has nothing that it has read but not processed, and if there is something on the flow named payment. This corresponds to Figure 11, because the meaning of that part of a P-spec is that the bubble is not allowed to fire concurrently with itself.

5.2 What Flows do Bubbles Read When Enabled?

When a bubble is enabled, it reads some of its input flows. Each consumable flow that is read is, of course, consumed, where consumption means removing the head of the sequence associated with the flow. However, for a persistent flow, the information on the flow is unchanged by the act of reading. (The information read from these flows is saved in the next configuration’s Read mapping by the firing rules below.)

The part of a P-spec that says what each enabled bubble will consume in a given configuration is captured in the curried, set-valued mapping:

\[ \text{Consume} : B \rightarrow (\text{FlowState} \rightarrow \text{PowerSet}(\text{PowerSet}(F))). \]

That is, for all bubbles \( b \), and all flow states, \( fs \), the set \( \text{Consume}(b)(fs) \) is a set of sets of flows. Because Consume is curried, for a bubble \( b \), \( \text{Consume}(b) \) is the mapping derived from \( b \)'s part of the P-spec. The mapping \( \text{Consume}(b) \) can also be thought of as a binary relation between flow states and sets of flows. A relation or set-valued mapping is needed to deal with possible nondeterminism in the P-spec.

Each set of flows in the set \( \text{Consume}(b)(fs) \) represents a possible set of flows that the bubble \( b \) may read when it is enabled in flowstate \( fs \).

In a valid formal model of a P-spec, the Consume mapping should only produce input flows for a bubble, should only depend on local information, should be coordinated with the Enabled function, and should not request that an empty flow be read.

The locality condition is that the flows read must be inputs to the bubble \( b \). That is, the following should be true for all \( b \in B \) and \( fs \in \text{FlowState} \):

\[
\text{fset} \in \text{Consume}(b)(fs) \Rightarrow \text{fset} \subseteq \text{Inputs}(b), \quad (4)
\]

The local dependency condition is that the \( \text{Consume}(b) \) mapping should only depend on the values read from the flows in \( \text{Inputs}(b) \), since those are the only values that it will
\[\text{Consume}_e(\text{Customers}) = \lambda fs. \{\\}\]
\[\text{Consume}_e(\text{Generate-Sales-Order}) = \lambda fs. \{\{\text{what}, f_{\text{custs}}\}\}\]
\[\text{Consume}_e(\text{Bill-Customer}) = \lambda fs. \{\{f_{\text{creditBC}}\}\}\]
\[\text{Consume}_e(\text{Keep-Customer-Accounts}) = \lambda fs.
\begin{array}{l}
\text{if } \neg \text{IsEmpty}(fs(f_{\text{cash}})) \text{ then } \{\{f_{\text{c-a-data}}, f_{\text{cash}}\}\} \text{ else } \{\\}\n\\
\text{or } \text{if } \neg \text{IsEmpty}(fs(f_{\text{internal}})) \text{ then } \{\{f_{\text{c-a-data}}, f_{\text{internal}}\}\} \text{ else } \{\\}\n\\
\text{or } \text{if } \neg \text{IsEmpty}(fs(f_{\text{creditKCA}})) \text{ then } \{\{f_{\text{c-a-data}}, f_{\text{creditKCA}}\}\} \text{ else } \{\\}\n\end{array}\]
\[\text{Consume}_e(\text{Collections}) = \lambda fs. \{\{\log, f_{\text{payment}}\}\}\]
\[\text{Consume}_e(\text{Mail}) = \lambda fs. \{\{f_{\text{invoice}}\}\}\]

Figure 13: An example Consume mapping, \(\text{Consume}_e\).

have access to in the operational semantics. Formally, this condition is stated as follows. For all \(b \in B, fs \in \text{FlowState}\), and \(fs' \in \text{FlowState}\):

\[(\forall f \in \text{Inputs}(b) : fs(f) = fs'(f)) \Rightarrow \text{Consume}(b)(fs) = \text{Consume}(b)(fs'). \quad (5)\]

The coordination condition is that the set of possible sets of flows read, \(\text{Consume}(b)(fs)\), is nonempty when \(\text{Enabled}(b)(fs, r(b))\) is true. That is, the following should be true for all \(b \in B\) and \(fs \in \text{FlowState}\):

\[(\text{Enabled}(b)(fs, r(b)) = \text{true}) \Rightarrow \text{Consume}(b)(fs) \neq \emptyset. \quad (6)\]

The sensitivity condition is that, for each possible set of flows to be read, \(fset \in \text{Consume}(b)(fs)\), and for each flow \(f \in fset\), there must be something on flow \(f\) in the flowstate \(fs\). That is, the following should be true for all \(b \in B\) and \(fs \in \text{FlowState}\):

\[(fset \in \text{Consume}(b)(fs)) \Rightarrow (\forall f \in fset, \neg \text{IsEmpty}(fs(f))). \quad (7)\]

As an example, the mapping \(\text{Consume}_e\), defined in Figure 13 would be suitable for the DFD of Figure 5. The clause for \(\text{Generate-Sales-Order}\) says that there is only one \(\text{FlowState}\) and \(\text{Read}\) pair possible, that only the flow named \(\text{what}\) is consumed (because the flow named \(\text{custs}\) is persistent), and that both of the input flows are read.

It is easy to check that the validity conditions are satisfied by \(\text{Consume}_e\) by cases, that is, bubble by bubble. The locality condition is easily checked by a quick inspection. For examples of checking the other conditions, consider the bubble \(\text{Customers}\), which is always enabled. Because it does not test the values of input flows, it trivially satisfies the local dependency condition. Because this bubble is always enabled, \(\text{Consume}_e(\text{Customers})\) always produces a non-empty set (the set containing the empty set), and thus satisfies the coordination condition. For this bubble, the sensitivity condition is trivially satisfied, because the only set in the set of sets of flows given by \(\text{Consume}_e(\text{Customers})\) is the empty set. The most interesting case is \(\text{Consume}_e(\text{Keep-Customer-Accounts})\). Because it only tests the values of its own input flows, it satisfies the local dependency condition. Checking its coordination condition involves checking that in each case where the bubble is enabled, that is when there is something on its persistent flow and at least one of its consumable flows, it returns a nonempty set. Sensibility for this case is a bit more tricky, but follows by comparison with the enabling condition.
\[ \text{Selected}_e(b) = \lambda(fs, sq) \cdot \{ g \mid \text{IsIn}(g, sq) \} \]

Figure 14: An example of Selected function, \( \text{Selected}_e \), where \( b \in B \).

5.3 When are Bubbles Ready to Produce Output?

Each bubble that has read something from its inputs that has not yet been used for output can produce outputs from one or more of these inputs. A P-spec for a bubble that can fire concurrently with itself has to have some way of describing two things: when the bubble can produce output from some input it has read, and if in that case, what inputs are to be used to produce the output. For a bubble that cannot fire concurrently with itself, the bubble would be allowed to produce output whenever it has read some input, the enabling rules would prevent a bubble in such a state from reading more inputs.

The part of a P-spec that captures what previously read inputs each bubble will select to produce output is captured by the following curried function:

\[ \text{Selected} : B \rightarrow ((\text{FlowState} \times \text{WhatRead}^*) \rightarrow \text{PowerSet(WhatRead)}) \]

That is, for all bubbles \( b \), and for all pairs of a FlowState mapping, \( fs \), and a sequence, \( sq \) of WhatRead mappings, \( \text{Selected}(b)(fs, sq) \) is a set of WhatRead mappings. Recall that a WhatRead mapping is how the model records what a bubble has read from its inputs; it maps flows to values. In the formal model, the inputs read by a bubble, but not yet used to produce output, are stored in a sequence of WhatRead mappings. (Recall that a sequence is used to make it easy to process the inputs in the order in which they were read, which is probably a common case.) The Selected function returns a set of WhatRead mappings; if the empty set is returned, this means that the bubble cannot produce any output, given its argument FlowState and sequence of WhatRead mappings.

A valid Selected\( (b)\) mapping should only depend on the states of the flows in Inputs\( (b)\), since those are the only flow states that it will be passed by the operational semantics. It should also be sensible in that the WhatRead mappings that it returns should all be in the sequence of WhatRead mappings that it is given as input.

Formally, the locality condition is stated as follows. For all \( fs \in \text{FlowState}, fs' \in \text{FlowState}, \) and \( sq \in \text{WhatRead}^*\):

\[ (\forall f \in \text{Inputs}(b) : fs(f) = fs'(f)) \Rightarrow \text{Selected}(b)(fs, sq) = \text{Selected}(b)(fs', sq). \]  \tag{8} \]

The sensibility condition is that, for all \( fs \in \text{FlowState} \) and \( sq \in \text{WhatRead}^*\):

\[ s \in \text{Selected}(b)(fs, sq) \Rightarrow \text{IsIn}(s, sq) \]  \tag{9} \]

An example of an Selected function for the DFD of Figure 5 is given in Figure 14. It acts trivially on all bubbles, \( b \). The value of Selected\( _e \) for each terminator is one way of imposing the minimal constraint on such a bubble. As the other bubbles are not allowed to fire concurrently with themselves, there will be at most one WhatRead function in their sequence that can be selected. It is trivial to check the validity conditions for such a selection function.
5.4 What do Bubbles Produce from Given Inputs?

A bubble that has selected some previously read inputs can be called upon to produce outputs for those inputs. The part of a P-spec that describes how each bubble produces outputs from given inputs is modeled by a mapping, Produce, which is similar to the Consume mapping above:

\[ \text{Produce} : B \rightarrow (\text{WhatRead} \rightarrow \text{PowerSet}((F \rightarrow \text{OBJECTS}_{\perp}))) \]

As usual, Produce is curried, so that for each bubble \( b \), \( \text{Produce}(b) \) is the is the mapping derived from \( b \)'s part of the P-spec. The idea is that for all bubbles \( b \), and all WhatRead mappings, \( s \), \( \text{Produce}(b)(s) \) is a set of mappings of type \( F \rightarrow \text{OBJECTS}_{\perp} \). Each of these mappings says what the bubble produces on each of its output flows. If the mapping returns \( \perp \) for a flow, that is taken as an indication that the bubble did not produce anything for that output flow.

In a valid formal model of a P-spec, the Produce mapping for should be coordinated with the Selected mapping, should only produce outputs for the output flows of a bubble, and should not depend on what might be read from flows that are not inputs to a bubble.

The coordination condition is that if for a given bubble, \( b \), flowstate, \( fs \), and sequence of WhatRead mappings, \( sq \), if a WhatRead mapping, \( s \), is one of those selected, then \( \text{Produce}(b)(s) \) should be nonempty. That is, the following should be true for all \( b \in B \), \( fs \in \text{FlowState} \), and \( sq \in \text{WhatRead}^* \):

\[ s \in \text{Selected}(b)(fs, sq) \Rightarrow \text{Produce}(b)(s) \neq \emptyset \quad (10) \]

The locality condition is that \( \text{Produce}(b)(s) \) should only produce outputs for \( b \)'s output flows. That is, the following should be true for all \( b \in B \) and all \( s \in \text{WhatRead} \) that are sensible (i.e., for all \( s \) such that \( \forall f \in F . f \notin \text{Inputs}(b) \Rightarrow s(f) = \perp \)):

\[ \forall f \in F . f \notin \text{Outputs}(b) \land s' \in \text{Produce}(b)(s) \Rightarrow s'(f) = \perp \quad (11) \]

The local dependency condition is that the \( \text{Produce}(b) \) mapping should only depend on the values read from the flows in \( \text{Inputs}(b) \), since those are the only values that it will have access to in the operational semantics. That is, the following should be true for all \( b \in B \), \( s \in \text{WhatRead} \), and \( s' \in \text{WhatRead} \):

\[ (\forall f \in \text{Inputs}(b). s(f) = s'(f)) \Rightarrow \text{Produce}(b)(s) = \text{Produce}(b)(s'), \quad (12) \]

To work an example, we use two auxiliary functions defined in Figure 15. These functions help construct maps of type \( F \rightarrow \text{OBJECTS}_{\perp} \). The function Out is used to define a mapping that represents putting a single output on one flow; the function AddOut is used to adding another output to such a mapping.

An example of a Produce mapping for the diagram in Figure 5 is given in Figure 16. For this example, assume that the functions needsCredit, makeCashOrder, makeCreditOrder, updateCashCust, updateCreditCust, makeBill, updateAccts, creditOfPayment, and updateLog are defined elsewhere (e.g., in a P-spec). Because the bubble Customers is a source, we know that it can produce output, but have no way of knowing what that output might be. Thus, \( \text{Produce}_{\perp}(\text{Customers}) \) allows that bubble to have any behavior that is consistent with the types of its outflows. The sink Mail is given a Produce mapping that produces no output.
Out : (OBJECTS × F) → (F → OBJECTS⊥)
Out(o, f) = λf'. if f' = f then o else ⊥)

AddOut : (OBJECTS × F) → ((F → OBJECTS⊥) → (F → OBJECTS⊥))
AddOut(o, f)(s) = λf''. if f'' = f then o else s(f'')

Figure 15: Auxiliary functions to help construct mappings of type F → OBJECTS⊥.

 Produce_e( Customers ) = λs.
   {Out(o, fwhat) | o ∈ Type Meaning( Need )}
   ⊔{Out(o, fpayment) | o ∈ Type Meaning( Money-and-Bills )}
   ⊔{AddOut(o1, fwhat)(Out(o2, fpayment)) | o1 ∈ Type Meaning( Need ), o2 ∈ Type Meaning( Money-and-Bills )}

 Produce_e( Generate-Sales-Order ) = λs.
   let ow = s(fwhat) in
   let oc = s(fcusts) in
   if needsCredit(ow, oc)
   then {AddOut(updateCreditCust(ow, oc), fcusts)(
         Out(makeCreditOrder(ow, oc), fcredit))}
   else {AddOut(updateCashCust(ow, oc), fcusts)(
         Out(makeCashOrder(ow, oc), fcash))}

 Produce_e( Bill-Customer ) = λs.
   let bl = makeBill(s(fcredit)) in
   {AddOut(bl, finvoice)(Out(bl, finvoice))}

 Produce_e( Keep-Customer-Accounts ) = λs.
   {Out(updateAccts(s), fc-a-data)}

 Produce_e( Collections ) = λs.
   let op = s fpayment in
   let ol = s flog in
   {AddOut(creditOfPayment(op), fcredit)(Out(updateLog(op, ol), flog))}

 Produce_e( Mail ) = λs. {λf. ⊥}

Figure 16: An example Produce mapping, Produce_e.
It is easy to check that the example mapping, Produce, only produces outputs on each bubble's output flows. Checking the coordination condition is trivial, because Produce,\(b\) is never empty, for any bubble \(b\). Checking that the local dependency condition is satisfied can be done by inspection of what inputs each bubble consults; the only difficulty in the example is that knowledge of the function \textit{updateAccounts} is needed to check Produce,\(\text{Keep-Customer-Accounts}\).

5.5 What is the Initial State of the Flows?

As discussed in Section 2.4, the final piece of information that needs to be specified in a P-spec is the initial flowstate, \(fs_{\text{initial}}\). For the example DFD of Figure 5, an appropriate \(fs_{\text{initial}}\) would be \(fs_e\) from Figure 10.

6 Operational Semantics for Firing

We now have enough machinery to define the semantics of firing rules.

6.1 Overview of the Semantics

From a given initial configuration, \(\gamma_0\), the transition rules described below may allow one to construct one or more possibly infinite sequences of configurations. Each sequence has the form

\[
\gamma_0 \rightarrow \gamma_1 \rightarrow \cdots \gamma_n \rightarrow \cdots
\]

if it is finite or

\[
\gamma_0 \rightarrow \gamma_1 \rightarrow \cdots \gamma_n \rightarrow \cdots
\]

otherwise. The symbol \(\rightarrow\) denotes a binary relation between configurations. It is technically described in the rules below, but informally represents one step in a DFD's computation. In a valid sequence, it must be the case that for each \(i\), \(\gamma_i \rightarrow \gamma_{i+1}\) is provable from the transition rules.

Since the initial flowstate, \(fs_{\text{initial}}\), for a DFD is given by its P-spec, the initial configuration, \(\gamma_0\) can be constructed from that and an initial \textit{Read} mapping as follows.

\[
\gamma_0 = (fs_{\text{initial}}, \lambda b. (\)) \quad (13)
\]

Given \(\gamma_0\), in general it is possible to construct many different, but valid, configuration sequences. The set of all such possible configuration sequences is the semantics of a DFD. It is a set because some of the bubbles may be nondeterministic, and because race conditions may result in different firings (as in Figure 1).

6.2 The Transition Rules

There are just two kinds of transitions allowed between configurations: an enabled bubble can read some of its inputs, and a bubble that has read some inputs and has selected some for use as outputs can produce output.

A few notes about the notation in the transition rules, which are given in Figure 17. Each rule (technically an axiom) consists of a transition between configurations and several side conditions. The side conditions are located to the right of the transition, and follow the word “where”. The side conditions must hold for the transition to take place [30].
\[(fs, r) \rightarrow (fs', r')\] where \(Enabled(b)(fs|_{Inputs(b)}, r(b)) = true,\)
\[fset \in Consume(b)(fs|_{Inputs(b)}),\]
\[r' = (\lambda b' . if b \neq b' then r(b') else Enq(r(b)),\]
\[\lambda f . if f \in fset then Head(fs(f))\]
\[\else \bot)),\]
\[fs' = (\lambda f . if f \notin fset then fs(f)\]
\[\else if Consumable(f) then Rest(fs(f)) else fs(f))\]
\[(fs, r) \rightarrow (fs', r')\] where \(\neg IsEmpty(r(b)),\)
\[s \in Selected(b)(fs|_{Inputs(b)}, r(b)),\]
\[s' \in Produce(b)(s),\]
\[r' = (\lambda b' . if b \neq b' then r(b') else Delete(r(b), s)),\]
\[fs' = (\lambda f . if f \notin Outputs(b) \lor s'(f) = \bot then fs(f)\]
\[\else if Consumable(f) then Enq(fs(f), s'(f))\]
\[\else Enq(\bot, s'(f)))\]

Figure 17: The transition rules.

All occurrences of each variable mentioned in a rule (both in the transition and the side condition) must have the same value, thus each use of \(b\) is the same. (That is, variables mentioned in a rule can be thought of as universally quantified outside the entire rule.) Finally, the notation \(fs|_{Inputs(b)}\) means the restriction of the mapping \(fs\) to the domain \(Inputs(b)\).

The first transition rule in Figure 17 states that if a bubble \(b\) is enabled, then it may consume the inputs in one of the sets of flows returned by applying \(Consume(b)\) to the state of \(b\)'s input flows. The particular choice of what inputs to consume from this set, denoted \(fset\) in the rule below, is arbitrary. The configuration is updated by recording what the bubble read in the new \(Read\) mapping, \(r'\), and by adjusting the flowstate by consuming each consumable flow that was read. The new flowstate in the updated configuration is denoted \(fs'\).

The second transition rule in Figure 17 states that if a bubble, \(b\), has at least one selected input (i.e., if \(Selected(b)(r(b))\) is nonempty), then one of these, \(s\), is chosen arbitrarily and used to produce a set of possible mappings \(Produce(b)(s)\). Each of these possible mappings has type \(F \rightarrow OBJECTS_{\bot}\), and represents what the bubble might produce on its output flows. One of these possible output mappings, \(s'\), is chosen arbitrarily and is used to produce the new flowstate \(fs'\). The first occurrence of the chosen input \(s\) is deleted from the read mapping for the bubble \(b\).

The sequence of configurations of Figure 18 is one of the possibilities for the first DFD of Figure 2 under these transition rules. In this example, the flow with name \(To\) is referred to as \(f_{To}\) and the flow with name \(From\) as \(f_{From}\).
\( (\lambda f \cdot \text{if } f = f_{\text{To}} \text{ then } \langle 2, 3 \rangle \text{ else } \langle \rangle, \lambda b \cdot \langle \rangle) \rightarrow \)

\( (\lambda f \cdot \text{if } f = f_{\text{To}} \text{ then } \langle 3 \rangle \text{ else } \langle \rangle, \lambda b \cdot \text{if } b = \text{H} \text{ then } (\lambda f \cdot \text{if } f = f_{\text{To}} \text{ then } 2 \text{ else } \bot) \\text{ else } \langle \rangle) \rightarrow \)

\( (\lambda f \cdot \langle \rangle, \lambda b \cdot \text{if } b = \text{H} \text{ then } (\lambda f \cdot \text{if } f = f_{\text{To}} \text{ then } 2 \text{ else } \bot) \\text{ else } \langle \rangle) \rightarrow \)

\( (\lambda f \cdot \text{if } f = f_{\text{From}} \text{ then } \langle 27 \rangle \text{ else } \langle \rangle, \lambda b \cdot \text{if } b = \text{H} \text{ then } (\lambda f \cdot \text{if } f = f_{\text{To}} \text{ then } 2 \text{ else } \bot) \\text{ else } \langle \rangle) \rightarrow \)

\( (\lambda f \cdot \text{if } f = f_{\text{From}} \text{ then } \langle 27, 4 \rangle \text{ else } \langle \rangle, \lambda b \cdot \langle \rangle) \)

Figure 18: An example configuration sequence for the first DFD of Figure 2.

7 Discussion

In this section we discuss related work and offer some conclusions.

7.1 Alternatives to Our Semantics

We now briefly consider some alternative semantics that are close to our semantics for our formal model of extended DFDs.

The semantics given does not allow any easy way for a bubble to go into an “infinite loop” when trying to produce output. One could imagine letting this be modeled by allowing \( \text{Produce}(b)\langle s \rangle \) to be an empty set. However, this is an unsatisfactory way to model such behavior, as then the bubble \( b \) cannot give the choice of either going into an infinite loop or doing something else. We leave a satisfactory solution to this problem as future work.

If one prefers not to specify the behavior of the sources and sinks of a DFD, then one could use a semantics that is a function from specifications (of the sources and sinks) to sets of sequences of configurations. This is easily done, but complicates the formal presentation of the semantics.

One might wish to abstract away from the sequences of configurations, in order to focus on the “answer” returned by a DFD. Indeed, this is often done in the semantics of programming languages given in Plotkin’s structural style [31] [30]. To do this, one would identify a set of terminal configurations, from which no transitions are possible. However, such an attempt seems of little value for DFDs, because the sources may always be able to fire, as in our example. Even if the sources stop sending outputs into the DFD, the rest of the DFD may be able to continue running. This would be normal in many applications.

A related idea (and a more interesting one for our purposes) is to extract a scenario (or scenario set) from a sequence of configurations. A method for translating from configuration sequences to scenario sets is necessary for studying refinement relationships. For finite computations, a scenario shows what values can be presented to the sinks of a DFD from given inputs and so does give the answer returned by the DFD. Infinite computations can be characterized with scenarios, although the simple graphical representation for scenarios used thus far is no longer sufficient.
The following algorithm for constructing scenario sets from configuration sequences is simplified and adapted for DFDs from Brock and Ackerman's algorithm for constructing the scenario set of a network computation [26]. The simplification results from the fact that Brock and Ackerman's algorithm finds all possible scenarios for a network computation, while the algorithm presented here finds only the scenarios arising from a given configuration sequence. Additionally, this algorithm deals only with finite configuration sequences. Brock and Ackerman's algorithm could be used to find all scenarios of a DFD, including those associated with infinite configuration sequences.

To construct a scenario (or scenario set) from a configuration sequence:

1. Define inputs and outputs of the DFD. One reasonable way to do this is to define any flow from a terminator as an input, and any flow to a terminator as an output.
2. Take any finite configuration sequence (or finite prefix of a configuration sequence).
3. Construct a table with one column for each flow of the DFD.
4. List all values that appear on a flow at some point in the configuration sequence in the column associated with that flow, in order of appearance on the flow.
5. For each value on an output flow of a bubble, identify the set of all values on input flows of the bubble used in producing that value. This can be done by examining the element of the WhatRead mapping used in the bubble's firing, and then finding the reading from input flows that added that element. In the table, draw arrows from each identified input flow value to the output flow value. If the bubble can fire concurrently with itself and has read the same set of inputs multiple times, replicate the entire table as many times as needed to represent each possible relationship of inputs to outputs. Repeat step 5 for each table so created.
6. In each column of the table, draw arrows from each value except the last to the next value.
7. Call each column associated with an input of the DFD an input column, and each column associated with an output of the DFD an output column. For each value in an input column, if there is a directed path from that value to a value in an output column, draw an arrow from the input column value to the output column value.
8. Remove any columns that are neither input nor output columns. Remove any arrows whose source or target values (or both) were removed.
9. From the remaining table, remove any arrows that are implied by transitivity.

### 7.2 Consumable Flows with Multiple Sources and Targets

Consumable flows, as well as persistent flows, are allowed to have multiple sources and targets in our DFDs. Although "merging" and "splitting" of flows has been studied by others, our semantics for a consumable flow with multiple sources and targets is neither merging nor splitting. Instead, if one studies the semantics, one sees that such a consumable flow acts like a synchronized message queue, allowing several clients to send messages, each of which is only picked up by one server. The two-step firing rule and the operational semantics force synchronization: at any transition to a new configuration, only one bubble
is active, and this bubble may either read or write, but not both. Hence, no simultaneous read or write is possible. This semantics has many potential applications. For example, one can model an unbounded job queue directly by connecting those bubbles requesting jobs as sources and those servicing jobs as targets to a single consumable flow.

However, this powerful semantics for consumable flows lets one write specifications that may be difficult to implement. If an extended DFD specification language permits the use of consumable flows with multiple sources and targets, then the implementer of a specification using consumable flows with more than one source and target must be aware of this semantics to ensure that the proper synchronization is implemented.

Whether an extended DFD specification language should allow such flows is a methodological point, and thus beyond the scope of this paper. We have chosen to make the semantic model more regular, and more expressive, by not requiring flows with multiple sources and targets to be persistent. It is up to a DFD specification language designer to decide if such flows are useful.

7.3 Related Work

We have already discussed some related work that describes ways to derive formal specifications from the products of SA, and which augments SA ERDs and DFDs with formal specifications. In this paper we have done neither of these things, although they are motivation for our work. Instead, we have defined a formal semantics for a “theoretical core variant of DFDs” that includes a way to model the dynamic behavior of such DFDs.

Our motivation for this work is to describe a translation target for extended DFD specification languages, and thus we have sought to produce a general and useful formal model. Aside from the work on high level timed Petri nets (HLTPNs) [32], we know of no other work with similar goals. The main distinction between our formal models and HLTPNs is that our models have a more direct correspondence to DFDs. HLTPNs have timing information associated with them, and so are permit timing analysis that is impossible with our models. We discuss work that uses HLTPNs further below.

In what follows we discuss work that is related in the sense that it defines various other notions of DFDs, including the original notions. This work is primarily interesting as a way of testing our claim that our formal model of DFDs can act as a translation target for many different extended DFD specification languages. To this end, as we discuss this work we try to point out the differences from our example DFDs, and how those features can be translated into our models. (Thus, as we mentioned in the introduction, such differences should not be seen as a defect in our work.) We also point out places where related work does not give a precise semantics, especially for dynamic behavior; this is only intended to show the value of our formalization, and how it extends traditional notions of DFDs.

In this part of the paper, “DFD” will no longer mean our theoretical core variant of DFD, but the traditional notion.

To summarize the discussion below, our formal model can serve as a translation target for a wide variety of extended DFD specification languages and notations, because in each case the features discussed can be translated into our formal model. Our formal model is also more expressive, in that it can easily describe the semantics of dynamic behavior, including bubbles firing concurrently with themselves, which allows a formal treatment of refinement.
7.3.1 De Marco and Yourdon

DFDs used in traditional structured analysis [1] [25, page 15] have a very informal flavor and some features that are not directly present in our models. For example, De Marco has graphical notations on flows for “conjunction” (\(*\)) and “disjunction” (\(\oplus\)) that are not a part of our formal model of the syntax of DFDs. Similarly, Yourdon makes graphical distinctions between “data” and “control” flows [4] [25, pages 27–28]. However, our formal model can encode this kind of information in a general way. Our model of P-specs allows one to say that a bubble consumes or produces on two flows at once, or one only; the advantage of our formal model is that it can also express any other computable way that bubbles could consume or produce on flows. Our formal model can express control flows in a variety of ways; for example, by using a persistent flow that transmits boolean values, or by using a consumable flow that transmits values of a one-element type (a signal).

De Marco’s DFDs feature converging and diverging flows [1]. However, their semantics is ambiguous. A converging flow may mean either that:

- there are several flows which all have their sinks in the same bubble, or
- several “elementary packets of data” are to be joined “to form a complex packet” (such as a tuple) [25, page 16].

In our model one can express the first by having several separate flows, with a flow having several sources as in Section 2.3. In our model, one can express the second meaning for converging flows by introducing a bubble to accept each of the “converging” flows and specifying how it combines them. This avoids any possible ambiguity. De Marco’s converging flows are ambiguous in the same way. In our model one can express such a diverging flow either as: several flows (one to each destination of a De Marco style diverging flow), with a flow having several targets as in Section 2.3, or with a flow into a bubble that splits the data into parts, and sends each part on a different output flow. Again our model does not suffer from the ambiguity inherent in De Marco’s DFDs. Ward, in [24] gives notation to disambiguate these two senses of converging and diverging flows, but leaves the process by which tupling and splitting of data is achieved implicit. Similarly, our model can represent “dialogue flows” by two separate flows.

A feature of De Marco’s DFDs that we do not model directly is the description of a system as a hierarchy of DFDs. However, our semantics does allow one to relate different levels of a hierarchical DFD, to see if the behavior of one level refines another. Because we allow bubbles to fire concurrently with themselves, we can give behavioral specifications to even the highest levels of a hierarchical DFD.

The only aspect of hierarchical DFDs that is not handled by our semantics is that the transition rules for configurations do not take the hierarchy into account (as in the semantics of hierarchical colored petri nets [33] [34]). However, by not doing so, we do not define the semantics of a higher-level bubble by a sub-DFD, but rather permit both the higher-level bubble and its sub-DFD to be given a behavioral specification. This permits abstract execution of a higher-level bubble, and a comparison with the behavior of the sub-DFD.

7.3.2 Ward

Ward [24] distinguishes discrete from continuous data flows; that is between discrete and analog data, which he defines as “a set of values defined continuously over a time interval”
Hatley and Pirbhai have a similar notion of continuity [35] [25, Section 5.3.2]. Since such data can only be modeled in a computer by discrete data, there seems to be no good reason to model this distinction. Instead, we have adopted Ward’s semantics for continuous flows as shared variables [24, page 203] as a feature of flows in our model. This feature of flows is orthogonal to the type of data on the flow. This is the distinction between consumable and persistent flows [25, Section 5.3.4].

The traditional view of stores is that they represent files [24, page 199]. Our model allows stores to have any type, including the type of a file.

Ward also includes in his extension of DFDs something called a “buffer”, which is highly ambiguous. He says that a “buffer is an abstraction on a stack or a queue” [24, page 200]. Stacks and (LIFO) queues certainly have different behavior, but both can be modeled in our semantics.

Ward gives a semantics of DFDs based “loosely on the execution of a Petri net” [24, pages 203–205]. However, his semantics are somewhat informal and ambiguous. For example, he does not clear up the ambiguity in the potential behavior of buffers. Ward does not discuss an initial marking of the DFD with tokens, and his semantics do not deal with the actual values of data on flows. Hence his semantics can only be approximate (a kind of abstract interpretation of the DFD), because the transitions between markings cannot depend on the values of data on the flows or the exact functions computed by bubbles. Our semantics can be regarded as a cleaned-up version of this idea.

7.3.3 Tse and Pong

Tse and Pong recognize that: “Transitions and places of Petri nets correspond, respectively, to processes and data flows of DFDs” [36, page 1]. They also give an algebraic model of DFDs. However, their semantics using Petri nets also ignores the values on the flows and so cannot describe the full behavior of a DFD.

Tse’s work [37] only deals with the syntax of DFDs, not their semantics.

7.3.4 Bruza and van der Weide

Bruza and van der Weide [38] give a semantics of DFDs in terms of extended Petri nets and path expressions. Like Tse and Pong, they model flows by places, and bubbles by transitions. They point out that the DFD notation is ambiguous and needs extra specification annotations to be unambiguous.

One similarity with our work is that they recognize the need for a two-step firing rule for bubbles. Like Tse and Pong, however, their semantics also ignores the values on the flows, and so cannot describe the full behavior of a DFD.

7.3.5 Colored Petri Nets

One can imagine giving a fuller account of the dynamic behavior of DFDs by extending Ward and Tse and Pong’s approach with colored Petri nets [39] [33]. (The different “colors” on tokens can stand for different values being passed in a DFD.) However, because the firing rules of Petri nets are atomic, each bubble in a DFD would have to be either modeled by a complex Petri net, or the semantics of a bubble’s firing would be atomic. This is because in a Petri net, a transition reads from its input places and writes its output places in an atomic step, and so using Tse and Pong’s idea would imply that a bubble in a DFD would
reads its input flows and write its output flows in an atomic step. However, as described in Section 2.5, such a semantics is inadequate for capturing the dynamic behavior of a DFD.

7.3.6 Data Flow Networks

Much of the research in the underlying semantics of distributed systems has been done in the context of data flow networks. This work has produced both denotational semantics, such as Kahn’s foundational work [40] and that of Staples and Nguyen [41], and operational semantics, such as that of Panangaden and Shanbhogue [42], for data flow networks. Any of these semantics that supports nondeterminism (Kahn’s does not) could be used as a translation target for DFDs, but the translation would present significant difficulties. For example, the computations that occur at the nodes of data flow networks are atomic, which complicates the translation of any DFD language in which bubbles do not fire atomically. Such DFD concepts as stores (and thus persistent flows), bubbles that fire concurrently with themselves, and flow convergence and divergence have no direct counterparts in data flow networks. Thus, any such translation would be indirect and likely cumbersome to use in practice.

However, many of the results from the study of data flow network semantics apply directly to DFDs. Of particular interest is the incompleteness of history relations for expressing the meaning of data flow networks, which translates directly to DFDs. As described in Section 2.6, we use scenarios to resolve this shortcoming in the same manner as Brock and Ackerman use scenarios to resolve this shortcoming for data flow networks [26].

7.3.7 Elmström, et al.

As part of the IPTES project, Elmström and others [19] [20] described a semantics for SA/RT DFDs [3] augmented with VDM-SL P-specs. This semantics uses high-level timed Petri nets (HLTPNs) [32]. HLTPNs are similar to colored Petri nets, but because they include timing information on transitions, they can adequately model the dynamic behavior of a DFD. This semantics has several features that are similar to our own semantic foundation for DFDs (although our work was done independently).

- The timing information associated with a transition in a HLTPN makes the semantics similar to our two-step firing rule. That is, firing a transition takes a certain amount of time, which allows race conditions to be modeled. The minimum and maximum times a transition may take to fire are an advantage in analyzing real-time systems.

- Elmström, et al. give a model of stores that is very similar to one of their models for a persistent data flow, which is thus similar to the way we model stores.

A minor difference from our work is that Elmström, et al. do not allow a separate specification of an initial state of a DFD’s flows; we argued in Section 2.4 above that in some cases this would make the specification of initialization very difficult. It would also make the model unsuitable as a general translation target for extended DFD specification languages.

A larger difference is that, in [19][20], Elmström, et al. do not handle hierarchical DFDs and refinement. Thus they do not deal with the possibility of bubbles firing concurrently with themselves.
The semantics given in [19] [20] is not as direct as our semantics, because a DFD specification is first translated into a HLTPN, and then the semantics of HLTPNs is given separately.

Finally, since they are only interested in one extended DFD specification language, Elmström, et al. have not produced a semantic translation target that is as general as our semantics. This is most evident in their presentation of P-specs, which are given in VDM-SL, instead of using a general semantic model.

### 7.3.8 Larsen, Plat, and Toetenel

Larsen, Plat, and Toetenel have given a formal semantics of DFDs by translation into VDM-SL [21]. In this work, the DFDs are augmented by VDM-SL P-Specs (mini-specifications), and the translation produces either an implicit style VDM-SL specification, or an explicit one (which can be executed). Again, this semantics is not as direct as our semantics.

In contrast to our work, their semantics treats DFDs as sequential processes. Thus their semantics is fundamentally different than ours. Their formal semantics also prohibits cyclic flows (which we use heavily). Furthermore, because they only deal with sequential systems, their semantics is more restrictive in what it can specify, because it enforces a one-to-one correspondence between inputs to a DFD and its outputs.

Also, since they are only interested in one extended DFD specification language, they do not give as general a translation target as our semantics.

### 7.3.9 Coleman

The static semantics of SA specifications (including notational issues) are treated in a dissertation by Coleman [25]. As noted above, we have drawn on this dissertation for its comprehensive discussion of the literature on DFDs and SA, for our model of flows as having both a name and a type, for the distinction between consumable and persistent flows and for the semantics of consumable and persistent flows. Coleman describes a notation for P-specs based on first-order logic [25, Chapter 7], which inspired parts of our model; his notation could be translated into our model.

Coleman also describes how one could give an operational semantics of SA specifications [25, Section 9.2] using what amounts to colored Petri nets (values are used instead of Ward’s tokens) and first-order logical assertions for the transition firing rules. As described above in the section on colored Petri nets, however, such a semantics is inadequate for modeling the dynamic behavior of DFDs. Nevertheless, our model is derived from his initial work.

Coleman also gives several different possible models of stores, one of which [25, Figure 8.2] is the one we adopt. However, he ignores stores in his sketch of the operational semantics of DFDs. Coleman does not treat a bubble firing concurrently with itself.

### 7.3.10 France

In the two works [43] [5] that we discuss below, France describes variants of DFDs, specification notations, and their semantics. Although his semantics are not general enough for our purposes, this is understandable, because his papers did not seek to give a general foundation for extended DFD specification languages; instead, his papers seek to give a semantics for the particular DFD variants and specification notations. As with the other work cited here, he is not as concerned with the dynamic behavior of DFDs as we are. In summary, the major difference is that France presents specific specification notations
and their semantics, while we have explored what kinds of semantics for extended DFD specification are sensible, with a special focus on modeling the dynamic behavior of DFDs.

**Semantically Extended DFDs** In [43] France works with DFDs that have several additional features, and he also gives a formal semantics. France’s “queued flows” are what we call consumable flows, and his “variables” are our persistent flows. France distinguishes two kinds of bubbles (data transforms and state transforms) and two kinds of flows (data and control). Our model is arguably simpler in that both of his types of bubbles and flows can be translated into ours.

France’s operational semantics of extended DFDs use a framework similar to ours: algebraic state transition systems (ASTSs) [44] [45]. The main difference in the formal framework is that France’s semantics use transition systems to model all parts of a DFD, not just the firing rules. This gives the semantics a nicely compositional flavor, and allows an easy treatment of hierarchical DFDs [43, page 333]. France gives an explicit notation for P-specs, which could be given an alternative semantics using our model. All of his additional graphical conventions for DFDs can be translated into our model in ways similar to those discussed above for other related work. France’s DFDs have syntactic and semantic restrictions that seem to be intended to enforce good methodology, but which also make them less general than ours. For example, each bubble must have an output flow (page 330) and nested loops are not allowed inside infinite loops in his P-spec statement language (page 337).

France treats stores the same as he treats other parts of a DFD, as ASTSs [43, pages 336–337]; hence his stores need more description than necessary in our model, where stores are modeled as persistent flows. However, France’s stores can be more powerful than this, as he points out that one can “specify that a subset of write actions have priority over a subset of read actions when they occur in parallel” (p. 337). Nevertheless, by translating such an active store into a bubble with a persistent flow to and from itself, such a store can be translated into our model. We believe that the translated version is somewhat better as a semantic basis, because it more clearly distinguishes active entities (bubbles) from passive ones (stores).

France’s notion of convergence and divergence is apparently the same as Ward and Mellor’s (see France’s Figure 2, p. 330). France calls these “binders” and “splitters”. Since he does not treat them in his formal semantics, his notions appear to suffer the same problems (no precise description of how the splitting or aggregation is done).

France’s model does not allow bubbles to fire concurrently with themselves because each bubble has only one copy of its variables [43, pages 344–345]. Firing in a DFD is apparently an atomic step, since “outputs are solely dependent on current inputs” [43, page 342]. Although France’s language for writing P-specs does include explicit statements that allow for parallel processing within a bubble, his conditions on such parallel processing [43, page 341] require that statements executed in parallel cannot interact. Thus the parallelism allowed within a bubble’s firing cannot be observed from the outside; from the outside a bubble appears to transform inputs to outputs in an atomic step. The reason for permitting such internal parallelism is that one can plan for, and more easily prove a refinement of a bubble into several bubbles; the idea is that each parallel process in a P-Spec can be replaced by a bubble in the refinement. Since the refined bubbles fire in atomic steps, the refinement cannot process inputs in a different order than the refined bubble, as noted in the I/O consistency condition [43, pages 342–343]. Although this is an advantage in that
it simplifies the proof that a refinement correctly implements the refined bubble, it is less expressive as a foundation for specification languages, because it prohibits some refinements. In particular, it would prohibit refinements in which the bubbles in the refinement are connected by flows, where these flows may affect the order of the execution of the bubbles. That is, there is no way to specify a bubble that may be refined into a system of bubbles that may process inputs in different orders. We permit such refinements by allowing a bubble to fire concurrently with itself.

A Predicative Basis for SA Specification Tools

In [5] France gives a different formalization of the semantics of DFDs, this one geared towards composition and decomposition of DFDs. In this work France uses a variant of DFDs he calls PDFDs, along with ERDs, and data dictionaries with ADT specifications. His PDFDs include terminators (external entities), as do ours.

In [5], the behavior of a bubble is specified with an input/output predicate, which can be represented in our model as the functions that tell what a bubble produces and consumes. However, his language is not able to express conditions on when a bubble is enabled.

The major difference between his semantics for a PDFD and our model is that his semantics regards a PDFD as an atomic data transformation. That is, his semantics says what outputs (and global state changes) a bubble or a PDFD can produce for a single input. Such data transformations are atomic, because they cannot model time delays and race conditions, as discussed in Section 2.5 above. While for some purposes this might be adequate to characterize the behavior of a single bubble (and for his example they seem adequate), it is inadequate as a general model for the dynamic behavior of entire DFDs, because it does not allow for concurrency within a DFD. For the same reason, France's composition operators are also inadequate as a basis for studying the dynamic behavior of DFDs: they operate on single inputs and generate an atomic data transformation. For example, the parallel composition operator does not allow one PDFD to run twice while the other waits.

Because of this atomic view of the PDFD semantics, France's notion of the correctness of a decomposition is too restrictive for a semantics of DFDs that is concerned with dynamic behavior, as described in Section 2.6 above.

7.3.11 Harel's Statecharts

While they are not intended as a semantic foundation for SA-style DFD specification languages, Harel's statecharts [47] are also a visual specification notation that allows for concurrent execution, and could be taken as an alternative translation target for DFD specification languages. However, the states in statecharts represent control and not data processing; thus there is no direct way to model the flows of a DFD in the framework of statecharts.

Our semantics solves the problems caused by mixing instantaneous events and those that occur over a span of time by using a two-step firing rule. In the case of statecharts, this problem (and other related problems) greatly complicate the operational semantics.

Hierarchy in statecharts is provided by states that contain statecharts. The behavior of such a superstate is defined in terms of the statechart it contains, so this kind of hierarchy does not introduce refinement in the sense that we discussed for DFDs.
7.4 Conclusions

In this paper we have described a formal foundation for the semantics of extended DFD specifications by giving a structural operational semantics of a theoretical core variant of DFDs. Our formalism should not be considered a proposal for notation that anyone would use in practice, as indeed we have tried to avoid describing P-spec and DFD notations in detail. Instead we have described semantics that can adequately capture the meanings of various extended DFD specification languages. In particular we have focused on capturing the dynamic behavior of DFDs.

The main problem with practical application of this work is to give a syntax for P-specs and to give its formal semantics using these ideas.

However, our work does have implications for practitioners and specification language designers. Our work on a precise formal semantics for DFDs provides the following practical insights.

- One can use an extended DFD specification language to specify the dynamic behavior of a system. These extensions involve adding the following information to DFDs:
  - when each bubble is enabled
  - what the bubble reads when it is enabled
  - what previously-read inputs the bubble selects for producing output,
  - what the bubble produces from a previously read input, and
  - the initial state of the flows in the system.

- In addition to flow names, one should put the types of data as labels on the flows, and specify these types as ADTs in the data dictionary. Having both the name and type on a flow prevents ambiguity and conveys more information. Allowing the types to be arbitrary ADTs permits modern software engineering practices to be used and helps keep the level of abstraction high.

- There is no fundamental difference between a control flow (or control bubble) and a data flow (or data bubble). Thus one can use data flows to achieve some measure of control, and designs without explicit control flows are not necessarily less worthy than those with explicit control flows.

- One can think of stores abstractly as shared variables, not just as files. Allowing a store to have an arbitrary abstract data type gives the design more flexibility and keeps implementation details out. (Of course, one useful ADT is a file, so no expressive power is lost.)

- One can think of a bubble as a collection of cooperating processes — not just as a procedure. This allows DFDs to be useful as high-level specifications for concurrent and distributed systems, and it keeps the DFD from becoming embroiled in low-level procedural details. The ADTs used by the DFD can (and will often) be implemented with procedures. Thinking of a bubble as a collection of cooperating processes recognizes that each bubble may be refined into another DFD. This is supported by our semantics which allows a bubble to fire concurrently with itself.

In some discussions we have heard it said that it is counter-productive to try to formalize DFDs: aren't DFDs supposed to be an informal notation that is intended to be understood
by customers? Wouldn't a formal notation be harder for a customer to understand? We believe that customers think they understand the DFD notation, but often understand something different than what was intended. We know that as an ambiguous, informal notation, DFDs are open to differing interpretations, sometimes with disastrous results. However, we do not want to stop people from using DFDs informally. What we want is to have the possibility to use some extension of DFDs in a precise, formal way. This precise use of DFDs would certainly come after the imprecise, informal use, but one should not have to completely change notations in order to formally describe systems. Instead, we look forward to an integration of informal and formal specifications spanning a wide range of needs [7] [16] [22]. The advantage of formal and precise DFD specifications would be that the work done in requirements analysis would not have to be thrown away when more precision is required.

Acknowledgements

Thanks to Kari Lyle for her work on refinement of DFDs, and her help with an earlier version of this paper. Thanks to Juergen Symanzik and two anonymous referees for helpful suggestions and corrections on an earlier version of this paper. Thanks to the seminar on Structured Analysis in Spring '91 (Joe Reynolds, John Rose, Bedky Wemhoff) and Spring '92 (Joe Reynolds, Bashar Jano, Soma Chaudhuri) for help in developing earlier versions of these formal models.

References


