Iowa State University
Digital Repository

Computer Science Technical Reports | Computer Science

11-15-1993

An Executable Semantics for a Formalized Data Flow Diagram Specification Language

Tim Wahls
Iowa State University

Albert L. Baker
Iowa State University

Gary T. Leavens
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Programming Languages and Compilers Commons, and the Systems Architecture Commons

Recommended Citation
Wahls, Tim; Baker, Albert L.; and Leavens, Gary T., "An Executable Semantics for a Formalized Data Flow Diagram Specification Language" (1993). *Computer Science Technical Reports.* Paper 160.
http://lib.dr.iastate.edu/cs_techreports/160

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

An Executable Semantics for a Formalized Data Flow Diagram Specification Language

TR93-27

Tim Wahls, Albert L. Baker, and Gary T. Leavens

November 15, 1993

Iowa State University of Science and Technology
Department of Computer Science
226 Atanasoff
Ames, IA 50011

An Executable Semantics for a Formalized Data Flow Diagram Specification Language

Tim Wahls, Albert L. Baker, and Gary T. Leavens

TR #93-27
November 1993

Categories and Subject Descriptors:
D.2.1 [**Software Engineering**] Requirements/Specifications — *languages*; D.2.2 [**Software Engineering**] Tools and Techniques — *computer-aided software engineering (CASE)*; D.2.m [**Software Engineering**] Miscellaneous — *rapid prototyping*
General Terms: Specification, Design, Prototyping, Formal Semantics
Additional Key Words and Phrases: Data Flow Diagrams (DFDs), Standard ML, executable specification, literate programming, operational semantics, specification language semantics, RT-SPECS, Structured Analysis (SA)

Submitted to ACM Transactions on Software Engineering and Methodology.
© Tim Wahls, 1993. All rights reserved.

Department of Computer Science
226 Atanasoff Hall
Iowa Sate University
Ames, Iowa 50011-1040, USA

An Executable Semantics for a Formalized Data Flow Diagram Specification Language

Tim Wahls[*], Albert L. Baker, and Gary T. Leavens
Iowa State University

November 15, 1993

Abstract

While traditional Data Flow Diagrams (DFDs) are popular, they lack the formality needed in a good specification technique. We provide an executable semantics for a subset of RT-SPECS, a formalization of DFDs, using the programming language Standard ML. RT-SPECS is a formal notation for specifying concurrent and real-time software that relies on model-based specification of abstract datatypes. Processes are specified using assertions rather than algorithms. Because our semantics of RT-SPECS is written in SML, it is also an interpreter, yielding a directly executable specification language.

Categories and Subject Descriptors:
D.2.1 [**Software Engineering**] Requirements/Specifications — *languages*; D.2.2 [**Software Engineering**] Tools and Techniques — *computer-aided software engineering (CASE)*; D.2.m [**Software Engineering**] Miscellaneous — *rapid prototyping*
General Terms: Specification, Design, Prototyping, Formal Semantics
Additional Key Words and Phrases: Data Flow Diagrams (DFDs), Standard ML, executable specification, literate programming, operational semantics, specification language semantics, RT-SPECS, Structured Analysis (SA)

1 Introduction

1.1 Data Flow Diagrams

Traditional Data Flow Diagrams (DFDs) are probably the most widely used specification technique in industry today. They are the cornerstone of the software development methodology commonly referred to as "Structured Analysis" (SA). Their popularity arises from their graphical representation and hierarchical structure, which allows users with non-technical backgrounds to understand them. Indeed, one of the common uses of DFDs is in explaining the static structure of a system to non-technicians.

[*]Wahls's work is supported in part by a research assistantship provided by the College of Liberal Arts and Sciences, Iowa State University. Baker's work was supported in part by a grant from Rockwell International. Leavens's work is supported in part by the National Science Foundation under Grant CCR-9108654.
Authors' address: Department of Computer Science, 226 Atanasoff Hall, Iowa State University, Ames, Iowa 50011-1040 USA

Traditional DFDs consist of bubbles and flows. Bubbles are drawn as circles and represent either processes (if the system being specified is concurrent) or procedures. Flows are drawn simply as arrows connecting the bubbles, and show the paths over which data may travel. Hence, a DFD is a directed graph. Flows coming into a bubble are called inflows, and flows leaving, outflows. A bubble reads the information on its inflows, and produces information on its outflows.

1.2 Motivation for formalization of traditional DFDs

While the description above is simplified, it is already enough to point out the greatest flaw of traditional DFDs — there is no way to know when a bubble will read its inflows or produce on its outflows, and there is no way to know what a bubble will produce. This is partially explained by the fact that traditional DFDs are intended as static "road maps" of how information is transformed in a system, so that there was no notion of actually "executing" a DFD. This is clearly insufficient for precisely specifying a real software system.

A number of techniques have been used in traditional DFDs to combat this problem. In particular, the functionality of a bubble is often expressed in "Structured English" [21]. This has the advantages of informality and some level of understandability by non-technicians, but is too ambiguous and low-level to be a good specification technique. Another technique is to include in the DFD the actual code implementing a bubble [3]. While this is a formal specification, it is not an adequate specification technique, as it is too low-level. A third modification to DFDs describes the functionality of bubbles with a finite state machine. This ends up as a notational convenience, for the finite state description is easily transformed into another traditional DFD.

A better way to augment traditional DFDs so that they can model software systems is RT-SPECS, a technique developed at Iowa State University (ISU). In RT-SPECS, a set of *rules* is associated with each bubble. Each rule has three parts: an *enabling condition* that describes when a bubble may read its inputs, a *pre-condition* that gives conditions that must be met for the bubble to produce results, and a *post-condition*, which defines what the bubble outputs to its outflows. These conditions are written as first order predicate calculus (FOPC) assertions over the values on the inflows and outflows of the bubble. Thus, RT-SPECS has much in common with SPECS (a non-graphical specification language developed at ISU), VDM [6], and Z [5] [18] [19] in that specification is done using FOPC pre- and post-conditions. RT-SPECS thus has the formality of these specification techniques, and also the advantage of a graphical notation.

1.3 Goal of this paper

The goal of this paper, then, is to provide a formal semantics for RT-SPECS. First we provide an informal syntax and semantics for RT-SPECS to introduce the reader to the language. Then we give a formal syntax and semantics for RT-SPECS, written in the functional programming language Standard ML [13]. This provides two distinct semantic views of RT-SPECS: a denotational view (if we think of the program as a function mapping an RT-SPECS specification to its meaning), and an operational view obtained by actually running the SML code. We have striven to provide sufficient explanation of the SML notation to allow the reader who is not fluent in SML to follow the formal development. As using full FOPC for the enabling, pre-, and post-conditions makes RT-SPECS undecideable, we subset the language by using only propositional logic for these conditions. As the purpose of this paper is to give a formal semantics for DFDs, and not to provide insights into logic programming, this is reasonable. We conclude with a discussion of of problems remaining in the area of formalizing traditional DFDs, and a description of the advantages of executable semantics/specification.

2 Informal Description of RT-SPECS

The description in this section follows that of Coleman [2]. The syntax of RT-SPECS is an extension of traditional DFD syntax. The semantics of RT-SPECS, however, is a more radical departure from traditional techniques, as the traditional DFD model does not have a well-defined semantics.

2.1 Informal Syntax of RT-SPECS

An RT-SPECS specification consists of a set of bubbles and a set of flows. Ordinarily, a Data Dictionary, which stores information about individual bubbles and flows, is also associated with each DFD. In this model, we attach the Data Dictionary information directly to the bubble or flow, and so Data Dictionaries are not discussed here.

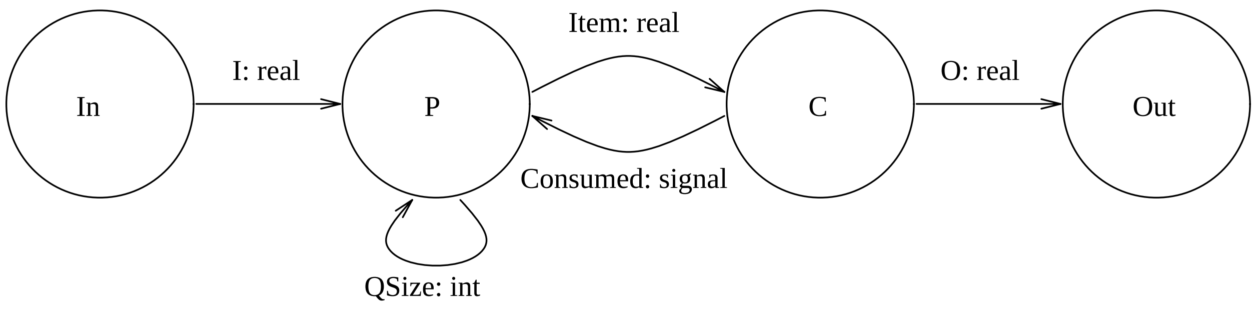
Each bubble is described by a *name* and a set of *firing rules*. The name is written inside the bubble on the diagram. The firing rules are the enabling condition, pre-, and post-condition triples discussed previously.

A flow is an arrow from one bubble to another labeled with a *name* and a *type*. The flow's name is just a string. The type gives the type of information (integer, string, ...) which may travel along the flow. In our model, the label "name: type" is required to be unique for each flow. There are two kinds of arrows in a DFD. Arrows with a double head are used for *persistent* flows. Those with single arrowheads are used for *consumable* flows. The difference between these kinds of flows is described in the next section.

To more precisely describe the non-graphical parts of an RT-SPECS specification, we give the following BNF grammar. The non-terminals *bubble-name*, *flow-name*, and *var-name* represent identifiers, and any non-terminal of the form *-literal* represents a constant of the given type. The symbol $\backslash$ is used for set difference, and $||$ for appending of sequences. The propositional logic subset of this concrete syntax includes all of the specialized RT-SPECS operators, and so the functionality of these operators will be explained along with the abstract syntax presented later.

textual-part ::= *process-list initial-state*
process-list ::= *empty* | *process-list process*
process ::= Process *bubble-name : rule-list*
rule-list ::= *rule* | *rule-list; rule*
rule ::= *enabling-condition* : *pre-condition* $\models$ *post-condition*
enabling-condition ::= true | *flow-enabled-list* $\wedge$ *FOPC-expr*
flow-enabled-list ::= *flow-enabled* | *flow-enabled-list* $\wedge$ *flow-enabled*
flow-enabled ::= $^{+}$*flow-name* | $^{-}$*flow-name*
initial-state ::= *empty* | Initial State: *flow-enabled-list*
$\mid$ Initial State: *flow-enabled-list* $\wedge$ *FOPC-expr*
pre-condition ::= *FOPC-expr*
post-condition ::= *FOPC-expr*
FOPC-expr ::= true | false | (*FOPC-expr*) | $\neg$*FOPC-expr* |
FOPC-expr $\wedge$ *FOPC-expr* | *FOPC-expr* $\vee$ *FOPC-expr* |
FOPC-expr $\Rightarrow$ *FOPC-expr* |
$\forall$*var-name* [*FOPC-expr*] |$\exists$*var-name* [*FOPC-expr*] |
token-expr rel-op token-expr | *token-expr set-op token-expr*
rel-op ::= $=$ | $\neq$ | $<$ | $\leq$ | $>$ | $\geq$
set-op ::= $\in$ | $\subset$ | $\subseteq$ | $\supset$ | $\supseteq$
token-expr ::= *int-literal* | *real-literal* | *string-literal* | *bool-literal* |
var-name | *flow-name* | *flow-name'*
unary-op(token-expr) | *token-expr binary-op token-expr* |
{*token-expr-list*}| <*token-expr-list*> |(*token-expr-list*)|
index(*token-expr, token-expr*)
token-expr-list ::= *empty* | *t-expr-list*
t-expr-list ::= *token-expr* | *t-expr-list, token-expr*
unary-op ::= size | first | header | last | trailer | length
binary-op ::= $+$| $=$ | $*$ |/| mod | $\cup$ | $\cap$ | $\backslash$ | $||$
empty ::=

As as example of an RT-SPECS specification, consider the following specification of a bounded buffer producer/consumer system from [1].



Process P:

$\quad {}^{+}I \wedge {}^{+}QSize \wedge {}^{-}Consumed \wedge QSize < 3:$

$\qquad \models QSize' = QSize + 1 \wedge Item' = I/2;$

$\quad {}^{+}I \wedge {}^{+}QSize \wedge {}^{+}Consumed:$

$\qquad \models QSize' = QSize \wedge Item' = I/2;$

$\quad {}^{-}I \wedge {}^{+}QSize \wedge {}^{+}Consumed:$

$\qquad \models QSize' = QSize - 1$

Process C:

$\quad {}^{+}Item : \models O' = Item + 1 \wedge Consumed' = ()$

Initial State:

$\quad {}^{-}Consumed \wedge {}^{-}Item \wedge {}^{-}O \wedge {}^{+}QSize \wedge QSize = 0$

The ${}^{+}flowname$ expression in the enabling condition is true exactly when there is information on the flow named $flowname$, and ${}^{-}flowname$ is true when there is no information on the indicated flow. Primed flow names ($'$) refer to outflows, while unprimed flow names are inflows. Bubbles In and Out represent the outside world, and so have no associated rules. Note that all flows in this example are consumable. A more detailed explaination of part of this example appears in the next section, but roughly bubble P produces reals onto flow Item, using the information on flow QSize to maintain the property that a maximum of three reals reside on Item at any time. The bubble C consumes reals from Item, and signals P that it has done so, using flow Consumed.

2.2 Informal Semantics of RT-SPECS

This informal description of RT-SPECS semantics follows the first formalization by Coleman [2] and the operational semantics given in Leavens et. al [8] for traditional DFDs. The key concept is that of *firing* a bubble. Firing is the process in which a bubble reads its inflows and produces onto its outflows. The semantics of firing gives meaning to the firing rules in RT-SPECS, which specify the dynamic behaviour of a DFD.

We model how a bubble fires in two steps. A bubble first reads its input flows, and then writes to its output flows. We say a bubble is *working* when it has read its input flows but not yet produced output; it is *idle* otherwise. We consider the transitions between these states to be atomic.

What happens when a flow is read depends on the persistency of the flow. When a bubble reads from a consumable flow, the information read is removed from the flow, while reading from a persistent flow does not affect the information it contains. Similarly, writing to a consumable flow adds to the information on the flow, while writing to a persistent one overwrites any information already present.[1] Thus, a persistent flow is like a variable shared between two

[1]The distinction between persistent and consumable flows is closely related to the issue of continuous versus discrete flows found in the traditional DFD literature [2] [4] [20].

bubbles, which only the origin bubble can write and only the destination bubble can read. We treat consumable flows as unbounded FIFO queues. Hence, any information read from a consumable flow is read from the head of the flow's queue, and any information added to a consumable flow is added at the rear.

Firing occurs as follows. Initially, all the bubbles are idle. The flows may have some initial values placed upon them — the input to the program being modeled. Then the following algorithm is executed:

1. Find the set of bubbles that may fire. This includes all bubbles in the working state, and any bubble in the idle state that has values on its inflows satisfying the enabling condition of at least one of its firing rules.
2. Choose one of these bubbles to fire.
3. Fire the bubble:
   - If the bubble is idle:
     (a) Choose one of the bubble's rules whose enabling condition is satisfied by the inflow values. The pre-condition is assumed to be true, as the bubble is not responsible for what happens if the pre-condition is false.
     (b) Read the values referenced by this rule from the inflows. If the inflow is consumable, remove the information from the flow. Otherwise, do not change the flow.
     (c) Change the state of the bubble from idle to working.
   - If the bubble is working:
     (a) Produce output onto the outflows. This output is defined by the post-condition of the rule chosen when the bubble changed to the working state. If the outflow is consumable, add the output to the flow. If the flow is persistent, overwrite whatever it currently contains.
     (b) Change the state of the bubble from working to idle.
4. Repeat the above steps until the set of bubbles allowed to fire in step one is empty.

So, in the example of the previous section, if we start in a state where all bubbles are idle, flow I has the value 2.0 at the head of its queue, flow QSize has the value 0 at the head of its queue, and no information is on flows Consumed and Item, then the only bubble which may fire is P, and the inflows only satisfy the enabling condition of its first rule. This rule has no pre-condition, which is equivalent to a pre-condition of just true. As both of P's inflows are consumable, the values mentioned above are read and removed from the flows. Finally, bubble P changes state to working.

If we fire the bubble again in the state resulting from the previous firing, then P is again the only bubble which may fire, and so (using the post-condition of its first rule) it enques the value 1 on flow QSize and 1.0 on flow Item. Finally, P changes state to idle.

3 RT-SPECS — in SML

While the above description gives useful intuition, those actually writing specifications in RT-SPECS need a more formal definition of the language. We now define the abstract syntax and semantics of RT-SPECS using the language Standard ML, which is noted for its similarity to standard denotational semantics notation. Thus, although SML is a programming language, it is also an effective tool for communicating with humans, and we view our work as equal parts executable specification and formal semantics of a specification language. See [11] for another example of this style of formal semantics.

3.1 Syntax

In this section, we give the abstract syntax for RT-SPECS in SML. We pay particular attention to parts of the syntax that are most needed by someone using the SML program to model a particular software system.

3.1.1 Flows

The syntax of flows is very close to that explained informally, so we give only the SML code and a small amount of explanation. Recall that a flow can be either persistent or consumable. The SML keyword `datatype` builds a discriminated union type whose constructors are listed on the right hand side of the equals sign. Hence, `Persistency` is a type with only two elements — `persistent` and `consumable`.[2]

⟨*flowtype declaration*⟩≡

```
  datatype Persistency = persistent | consumable;
```

In the following SML record implementation of flows, the `Origin` and `Destination` fields are the names of the initial and terminal bubbles of the flow, `Name` gives the name of the flow, and `Pers` records whether the flow is persistent or consumable.

The `Type` field is used to indicate the type of information that may travel on the last field of a flow, `Contents`. Borrowing from the language of Petri nets [14], we refer to each item of information as a token. The string stored in the `Type` field of a flow must be written according to the following grammar. This grammar renders types with parenthesized expressions, so that a flow that is meant to carry tokens such as `{<1>}` must have its type expressed as `(set of (sequence of int))`. *FieldID* is any nonempty sequence of alphanumeric characters.

Tokentype ::= int | real | bool | string | signal | (set of *Tokentype*)
(sequence of *Tokentype*) | (tuple (*Fieldexprlist*))

Fieldexprlist ::= (*FieldID*: *Tokentype*) | (*FieldID*: *Tokentype*), *Fieldexprlist*

The `Contents` field of a flow contains the tokens actually on the flow, and is modeled as a FIFO queue. As only one data item may exist on a persistent flow at a time, the `Contents` field of a persistent flow is always a queue of length zero or one. The natural data structure for building queues is a sequence, and so the contents field is a sequence of tokens. But this corresponds exactly to our notion of a token of type sequence, and so the `Contents` field is itself a token. Thus, flows are actually implemented (not just modeled) as a sequence of tokens.[3]

⟨*flow declaration*⟩≡

```
  structure Flow : FLOW =
    struct
    datatype F = Flow of {Origin:string,
                          Destination:string,
                          Name:string,
                          Type:string,
                          Pers:Persistency,
                          Contents:Token.T}
  ⟨omitted flow implementation⟩
  end;
```

[2]All SML code in this paper is presented using the literate programming tool Noweb [16]. Expressions of the form ⟨*decl*⟩ represent parts of the code that are presented elsewhere. The corresponding form ⟨*decl*⟩ ≡ `code` defines such omitted code. The advantages of Noweb are that it allows a program and the text describing it to be built simultaneously and in the same file, and that it provides a structured way of presenting the code.

[3]The `structure` keyword in SML introduces the body of an ADT, much like an Ada package body, while the associated `signature FLOW` (not shown) specifies client access to the ADT, much like an Ada package specification.

The only operation needed on flows at this level is the ability to build one. This is given by the constructor `makeFlow`, whose four string arguments are: origin name, destination name, flow name, and type — the type of token that may travel on the flow. Recall that is this context, `F` is the type of flows.

⟨*flow constructor*⟩≡

```
   val makeFlow: string * string * string * string * Persistency -> F
```

3.1.2 Bubbles

The other components of an RT-SPECS specification are the bubbles. In what follows we describe the structure of bubbles — while attempting to keep a clear distinction between our model of bubbles, and any additional information needed in our SML implementation.

Recall that a bubble may be either working or idle.

⟨*bubble state declaration*⟩≡

```
  datatype StateType = working | idle;
```

In the bubble record type defined below, the `Name` field is a unique identifier for a bubble, while `State` captures whether a bubble is working or idle. The `Rules` field gives the list of firing rules associated with the bubble. Each rule has a fairly complex structure and so we discuss rules after the description of bubbles.

The rest of the fields of a `Bubble` are used in our implementation, but are not part of the RT-SPECS model. For example, the `InFlows` and `OutFlows` fields just store the names and types of all inflows to the bubble, and all outflows from the bubble. This information is already present in the specification, but is difficult to access. The `Envmt` and `CurRule` fields are used when the bubble fires — the first holds the current environment being used to evaluate the rules, and the second caches the rule that a bubble is using to fire while it is in the working state. These steps are explained in greater detail in the semantics section that follows.

⟨*bubble declaration*⟩≡

```
  structure Bubble : BUBBLE =
    struct
    datatype B = Bubble of {Name:string,
                             State:StateType,
                             Rules:Rule list,
                             InFlows:(string * string) list,
                             OutFlows:(string * string) list,
                             Envmt: Env.E,
                             CurRule:Rule};
    ⟨omitted bubble implementation⟩
  end;
```

The syntactic interface used by the rest of the semantics for bubbles is described in the following signature. The type “`B`” is used for “bubble” in the signature.

⟨*bubble signature*⟩≡
```
  signature BUBBLE =
    sig
    type B
    ⟨bubble exceptions⟩
    ⟨bubble operations⟩
    ⟨idle/working operations⟩
    ⟨omitted bubble operations⟩
  end;
```

The operations used to construct bubbles are: `makeBubble`, which constructs a new bubble with name "string", and `addRule`, which is used to add a rule to the rule list of the argument bubble. These allow a bubble to be constructed incrementally. These operations are not used once we start interpretation (execution) of a DFD.

⟨*bubble operations*⟩≡
```
   val makeBubble: string -> B
   val addRule: B * Rule -> B
```

The following operations are provided for testing and changing the state of a bubble[4] — transforming it from idle to working, and back again. Operation `makeWorking` takes a bubble, a rule, and a list of flow name and token pairs. The list contains precisely those flows and corresponding token values that the bubble consumes when making the transition to working. These are used to construct the environment in which the current rule (the parameter of type Rule in the call to `makeWorking`) will be evaluated.

⟨*idle/working operations*⟩≡
```
    val isIdle: B -> bool
    val makeIdle: B -> B
    val isWorking: B -> bool
    val makeWorking: B * Rule * (string * Token.T) list -> B
```

3.1.3 Rules

What remains in our definition of the syntax of RT-SPECS is to give the structure of rules. A rule is a triple consisting of an enabling condition, a pre-condition, and a post-condition. The enabling condition resembles the *when* clause of GCIL [9].

⟨*rule declaration*⟩≡
```
  type Rule = (enabletype * preexpr * postexpr);
```

Each of the three conditions is a propositional logic expression written over RT-SPECS types. We now give the SML representations for these expressions, and a brief explanation.

We start with `tokenexprs`, which are token valued expressions. They are the building blocks for the enabling, pre-, and post-conditions — they will be combined with another type of expression to produce propositional logic assertions.

[4]When we say "changing the state of a bubble", we are speaking loosely of "modeling the changing state of the bubble."

⟨*tokenexpr declaration*⟩≡
```
datatype tokenexpr =
   ⟨primitive declarations⟩
   ⟨flow referencing declaration⟩
   ⟨token operation declarations⟩
;
```

The primitive `tokenexpr`s mark constants appearing in the rules.

⟨*primitive declarations*⟩≡
```
                          primi of int
                        | primr of real
                        | prims of string
                        | primb of bool
```

Tokens read from one of the bubble's inflows are accessed by the `tokenexpr inflow`. Thus, for example, the expression `inflow("foo")` refers to the token read from inflow `"foo"`, and can return any possible type of token — depending on the type of flow `"foo"`.

⟨*flow referencing declaration*⟩≡
```
                        | inflow of string
```

The token operation declarations give the syntax of all operators on tokens that return tokens as their result. Boolean valued operators, such as the relational operators on numbers, subset, and membership for sets and sequences are logical assertion constructors, and so are discussed with the enabling, pre-, and post-condition syntax. The types for tokens that we currently support are taken from the model based specification language SPECS. Thus, the possible types for tokens are: set, sequence, tuple, integer, real, string (of characters), and signal. The actual implementation of tokens and the operations on them is omitted, as it is just an exercise in SML programming with no real bearing on the ideas presented in this paper. However, we will describe the functionality of the operators informally.

⟨*token operation declarations*⟩≡
```
      ⟨numerical operators⟩
      ⟨set operators⟩
      ⟨sequence operators⟩
      ⟨tuple operators⟩
      ⟨boolean operators⟩
```

We provide the standard operations on numbers. All may be used with integers or reals, except that modulo works only on integers. Both arguments must be of the same type — adding a real and an integer is not permitted.

⟨*numerical operators*⟩≡
```
                        | add of (tokenexpr * tokenexpr)
                        | sub of (tokenexpr * tokenexpr)
                        | divide of (tokenexpr * tokenexpr)
                        | mult of (tokenexpr * tokenexpr)
                        | modulo of (tokenexpr * tokenexpr)
```

The operators on sets include the usual set operators, as well as operator `buildset`, which converts an SML list of tokens to a set containing the list elements. We require sets to be homogenous, so each of `union`, `intersection`, and `difference` require that both argument sets contain the same type of tokens. Operator `size` returns the cardinality of its argument set.

⟨*set operators*⟩≡
```
                    | buildset of (tokenexpr list)
                    | union of (tokenexpr * tokenexpr)
                    | intersection of (tokenexpr * tokenexpr)
                    | difference of (tokenexpr * tokenexpr)
                    | size of tokenexpr
```

The operators on sequences are similar to the operators on lists usually provided in functional languages. Operator `buildseq` converts an SML list of tokens to a sequence, much as `buildset` does for sets. Sequences are also homogeneous, so operator `append`, which concatenates two sequences, requires that its arguments contain the same type of tokens. Operator `first` returns the first element of a sequence, while `header` returns a sequence containing all the elements of its argument, except the last one. Thus, `header`$(< a, b, c >) =< a, b >$. Similarly, `last` returns the last element of a sequence, and `trailer` returns a sequence containing every element of the argument sequence except the first one. Thus, `trailer`$(< a, b, c >) =< b, c >$. Operator `length` returns the size of a sequence, while `index` takes a sequence and an integer n, and returns the nth element of the sequence.

⟨*sequence operators*⟩≡
```
                    | buildseq of (tokenexpr list)
                    | append of (tokenexpr * tokenexpr)
                    | first of tokenexpr
                    | header of tokenexpr
                    | last of tokenexpr
                    | trailer of tokenexpr
                    | length of tokenexpr
                    | index of (tokenexpr * int)
```

Operation `buildtuple` takes a list of pairs, where each pair is a field name and a `tokenexpr`, and returns a tuple with each field associated with the value of the paired `tokenexpr`. If `buildtuple`'s argument is an empty list, then it returns an empty tuple, which is the only member of type signal. Having such a type is useful in specifications when the only information required from a flow is the existence or nonexistence of a token. The flow Consumed in the producer/consumer bounded buffer specification is an example.

The `field` operator takes a field name and a tuple, and returns the token associated with that name in the tuple.

⟨*tuple operators*⟩≡
```
                    | buildtuple of ((string * tokenexpr) list)
                    | field of (string * tokenexpr)
```

The boolean operators take and return boolean valued tokens. The "`t`" prefix is added to distinguish them from SML's built in boolean operators.

⟨*boolean operators*⟩≡
```
                    | tand of (tokenexpr * tokenexpr)
                    | tor of (tokenexpr * tokenexpr)
                    | timplies of (tokenexpr * tokenexpr)
                    | tnot of tokenexpr
```

Thus, the `datatype tokenexpr` allows the building of arbitrarily complex token valued expressions. For example,

```
 index(buildseq [primi 3, primi 4], add(primi 1, primi 1))
```

would evaluate to `primi 4`.

Now that we have syntax for dealing with tokens, we can give the syntax for rules, as they are propositional logic assertions over tokens. The first component of a rule is the enabling condition. An enabling condition has two parts, the first of which is a list of flow names tagged with a `plus` or `minus`, and the second a `preexpr`, which represents a logical assertion exactly like the next component of a rule — the pre-condition. A rule may also have a trivial enabling condition — always true, denoted `T`.

In the `flowenabled list` section of an enabling condition, a `plus` tag on the name of the inflow means that a token is present on that flow, while the `minus` tag indicates that a token is not present on that flow. Any inflow name appearing anywhere else in the rule must appear as a `plus` item in the `flowenabled list`. Interestingly, a list is sufficient structure here — there is only one reasonable way for the `flowenabled` components to be related. This is a dramatic simplification of other efforts to capture this notion [7] [2], and is discussed further in the conclusions section.

⟨*enabling condition declarations*⟩≡
```
  datatype flowenabled = plus of string | minus of string;
  datatype enabletype = T | flowlist of ((flowenabled list) * preexpr);
```

The second component of a rule (the pre-condition) has the same syntax as the second component of the enabling condition. A `preexpr` is a propositional logic assertion over token expressions. We also allow boolean constants (hence the `prep` component) and the use of boolean tokens (from the `prebool` component) in these assertions. The standard propositional logic connectives provide more building blocks for assertions. The constructors `preless` through `pregreatereq` represent the obvious relational operators, and may be applied to `tokenexpr`s that evaluate to integers, reals, strings, or characters. The constructors `preeq` and `preneq` may be applied to any type of token, but as with the other relational operators, require that both argument `tokenexpr`s represent expressions that are of the same type. Constructor `presubset` requires that its arguments evaluate to sets of the same type of tokens, and `premember` requires that its second argument be a set or sequence, and that the type of its first argument be the element type of that set or sequence.

⟨*pre-condition declaration*⟩≡
```
  datatype preexpr = prep of bool
                   | prebool of tokenexpr
                   | preneg of preexpr
                   | preand of (preexpr * preexpr)
                   | preor of (preexpr * preexpr)
                   | preimplies of (preexpr * preexpr)
                   | preless of (tokenexpr * tokenexpr)
                   | prelesseq of (tokenexpr * tokenexpr)
                   | pregreater of (tokenexpr * tokenexpr)
                   | pregreatereq of (tokenexpr * tokenexpr)
                   | preeq of (tokenexpr * tokenexpr)
                   | preneq of (tokenexpr * tokenexpr)
                   | presubset of (tokenexpr * tokenexpr)
                   | premember of (tokenexpr * tokenexpr);
```

Syntactically and semantically, post-conditions are nearly identical to pre-conditions. The only differences are the substitution of `post` for `pre` (to satisfy SML's requirements that the names of datatype constructors be unique), and the addition of `postassign`, which is used to put a token, represented by the `tokenexpr` argument, out on the outflow with name `string`.

⟨*post-condition declaration*⟩≡
```
datatype postexpr = postp of bool
                  | postbool of tokenexpr
                  | postneg of postexpr
                  | postand of (postexpr * postexpr)
                  | postor of (postexpr * postexpr)
                  | postimplies of (postexpr * postexpr)
                  | postless of (tokenexpr * tokenexpr)
                  | postlesseq of (tokenexpr * tokenexpr)
                  | postgreater of (tokenexpr * tokenexpr)
                  | postgreatereq of (tokenexpr * tokenexpr)
                  | posteq of (tokenexpr * tokenexpr)
                  | postneq of (tokenexpr * tokenexpr)
                  | postsubset of (tokenexpr * tokenexpr)
                  | postmember of (tokenexpr * tokenexpr)
                  | postassign of (string * tokenexpr);
```

For example, here is the SML rendition of the first rule for bubble P in the earlier producer/consumer example:

⟨*rule example*⟩≡
```
      (flowlist([plus("I"), plus("QSize"), minus("Consumed")],
       preless(inflow("QSize"), primi 3)),
       prep true,
       postand(postassign("QSize", add(inflow("QSize"), primi 1)),
               postassign("Item", divide(inflow("I"), primr 2.0)))))
```

which states that:

- enabling condition: tokens must exist on inflows `I` and `QSize`, but not on inflow `Consumed`, and the value of the token on `QSize` is less than 3.
- pre-condition: always true
- post-condition: outflow `QSize` receives the integer read from it plus one, and outflow `Item` receives the real read from inflow `I` divided by 2.0. Note that it is possible for a flow to be an inflow to and outflow from the same bubble.

We will not expect specifiers to write expressions in this form. The reader should think of this example as a parse of the more natural syntax given in Section 2. Recall that our goal for this paper is to present an executable semantics for RT-SPECS, and so the syntax presented here is intended only as a basis for that semantics.

3.1.4 The RT-SPECS Model

Now that we have built all the pieces of the RT-SPECS model, we are ready to assemble them as complete, "formalized" DFDs. (Technically, the model also includes the Data Dictionary. We omit it here for the reasons previously discussed.) We have stated that the diagram consists of a set of flows and a set of bubbles.

⟨*Dfd declaration*⟩≡
```
structure Dfd: DFD =
 struct
 type D = {Flows: FlowSet.FSet, Bubbles: BubbleSet.BSet};
 ⟨omitted Dfd implementation⟩
end;
```

We have omitted the description of the structures `FlowSet` and `BubbleSet` as they are uninteresting — both sets are implemented by SML lists.

Next we provide the signatures of operations on `DFD`s and a brief description of their functionality.

⟨*Dfd signature*⟩≡
```
  signature DFD =
    sig
    type D

    ⟨Dfd exceptions⟩
    ⟨the empty Dfd⟩
    ⟨Dfd construction operations⟩
    ⟨fire declaration⟩
    ⟨firerule declaration⟩
    ⟨Meaning declaration⟩
    ⟨Run declaration⟩
    ⟨RandRun declaration⟩
    ⟨runaux declaration⟩
  end;
```

Building a `Dfd` starts from `empty`, which is just the value of the empty diagram — an empty set of flows and an empty set of bubbles.

⟨*the empty Dfd*⟩≡
```
   val empty: D
```

To make interesting diagrams, one uses the following operations. The operation `addBubble` adds a bubble constructed by `Bubble.makeBubble` to the `Dfd`. The information about inflows and outflows is added to the bubble when the appropriate flows are added. Hence, there is an ordering constraint here, as a flow's origin and destination bubbles must both exist before it can be created. Operation `addFlow` adds a new flow (constructed by `Flow.makeFlow`) to the `Dfd`. Operations `putToken` and `currToken` work on the flow specified by the name and type given in their second and third arguments; `putToken` places a token on the named flow in the `Dfd`. The type of the token is checked against the type of the flow. The token is placed at the end of the flow, so `putToken` enqueues the token. Operation `currToken` returns the token currently at the head of the flow.

⟨*Dfd construction operations*⟩≡
```
   val addBubble: D * Bubble.B -> D
   val addFlow: D * Flow.F -> D
   val putToken: D * string * string * Token.T -> D
   val currToken: D * string * string -> Token.T
```

As an example of using these specification constructor operations, here is the SML version of the producer/consumer problem specification presented in the second section. For purposes of this example only, the SML syntax `val varname = ...` may be thought of as assigning a value to `varname`, even though it isn't actually an assignment in SML.

⟨*producer/consumer problem*⟩≡
```
(* the producer/consumer bounded buffer *)
local open Dfd in

  val d = empty;
  ⟨bubble construction⟩
  ⟨flow construction⟩
  ⟨state initialization⟩
end;
```

First, construct the bubbles and add the necessary rules, using `makeBubble` and `addRule`.

⟨*bubble construction*⟩≡
```
val b = Bubble.makeBubble("Out");
val d = addBubble(d, b);
val b = Bubble.makeBubble("C");
val b = Bubble.addRule(b, (
          flowlist([plus("Item")], prep true),
          prep true,
          postand(postassign("O", add(inflow("Item"), primr 1.0)),
                  postassign("Consumed", buildtuple([])))));
val d = addBubble(d, b);
val b = Bubble.makeBubble("P");
val b = Bubble.addRule(b, (
          flowlist([minus("I"), plus("QSize"), plus("Consumed")], prep true),
          prep true,
          postassign("QSize", sub(inflow("QSize"), primi 1))));
val b = Bubble.addRule(b, (
          flowlist([plus("I"), plus("QSize"), plus("Consumed")], prep true),
          prep true,
          postand(postassign("QSize", inflow("QSize")),
                  postassign("Item", divide(inflow("I"), primr 2.0)))));
val b = Bubble.addRule(b, (
          flowlist([plus("I"), plus("QSize"), minus("Consumed")],
                   preless(inflow("QSize"), primi 3)),
          prep true,
          postand(postassign("QSize", add(inflow("QSize"), primi 1)),
                  postassign("Item", divide(inflow("I"), primr 2.0)))));
val d = addBubble(d, b);
val b = Bubble.makeBubble("In");
val d = addBubble(d, b);
```

Then, construct and add the necessary flows with `makeFlow` and `addFlow`.

⟨*flow construction*⟩≡
```
val d = addFlow(d, Flow.makeFlow("C", "Out", "O", "real", consumable));
val d = addFlow(d, Flow.makeFlow("C", "P", "Consumed", "signal", consumable));
val d = addFlow(d, Flow.makeFlow("P", "C", "Item", "real", consumable));
val d = addFlow(d, Flow.makeFlow("P", "P", "QSize", "int", consumable));
val d = addFlow(d, Flow.makeFlow("In", "P", "I", "real", consumable));
```

Finally, initialize the state of the diagram, using `putToken` to place the specification's input on the appropriate flows.

⟨*state initialization*⟩≡

```
   val d = putToken(d, "QSize", "int", Token.inttoken 0);
   val d = putToken(d, "I", "real", Token.realtoken 2.3);
```

3.2 Semantics

The example just given is a specification for a software system. To understand what this specification means, we need to understand the semantics of RT-SPECS — which means that we have to understand what it means to fire a bubble, and how bubbles and rules are selected for firing. To do this, we first formally define the meaning of an RT-SPECS specification by showing the appropriate SML code, and then name and informally explain some of the functions that allow a user to experiment with a specification.

The operation `Meaning` takes a diagram and fires all possible bubbles and rules in all possible combinations, returning the set of final configurations thus produced. `Set` is a generic set implementation. By final configurations, we mean diagrams in which no rule may fire. Returning a set of final configurations is necessary because an RT-SPECS DFD may not determine a unique final configuration, as at any stage there may be several bubbles enabled, and several rules enabled in each bubble. One can view `Meaning` more abstractly — as a semantic function that maps an RT-SPECS specification to its meaning, which is the set of final configurations. Thus, we have a denotational semantics along with the operational semantics obtained by viewing the SML code as an RT-SPECS interpreter.

⟨*Meaning declaration*⟩≡

```
   val Meaning: D -> D Set.S
```

Function `BubbleSet.BList` takes a `BubbleSet` and returns the bubbles in it in an SML list, thus allowing function `meanaux` to recurse through the bubbles one at a time.

⟨*function Meaning*⟩≡

```
   fun Meaning({Flows=f, Bubbles=b}) =
     meanaux({Flows=f, Bubbles=b} , BubbleSet.BList(b), false);
```

If we view the execution of `Meaning` as computing a tree of diagram configurations, then the return value, if any, is the set of leaves of the tree. Function `meanaux`, of type `Dfd.D * Bubble.B list * bool -> Dfd.D Set.S`, fires all the bubbles of the diagram, and so provides the "breadth" of the tree, while function `onebubble`, of type `Dfd.D * Bubble.B * Rule list -> (Dfd.D Set.S * bool)` computes the result of firing one bubble, and so traverses the "depth". Function `Bubble.BRules` returns the list of rules associated with its argument bubble. If no rules in the bubble have a true enabling condition (i.e. when the variable `found` is false after traversing the entire diagram), `meanaux` returns that configuration of the diagram as a part of the result.[5]

[5]SML functions employ a form of pattern matching something like that of Prolog. Each repetition of the function's name following a | starts a new "clause" of the function's definition. The SML compiler matches the actual arguments to the function against the patterns provided in the formal argument positions for each "clause", the first that matches is selected, and the body of that alternative is executed. Underscores (_) represent "wildcards" in the pattern, and so match any actual argument.

⟨*functions onebubble and meanaux*⟩≡
```
   fun onebubble(dfd, _, []) = (Set.Emptyset, false)
     | onebubble({Flows=f, Bubbles=b}, bub, r::rs) =
       if not(Bubble.isWorking(bub)) then
         ⟨firing an idle bubble⟩
       else
         ⟨firing a working bubble⟩
   and meanaux(dfd, [], found) = if found then Set.Emptyset
                                          else Set.Mkset(dfd)
     | meanaux(dfd, b::bs, found) =
       let val result = onebubble(dfd, b, Bubble.BRules(b)) in
         Set.Union(fst(result), meanaux(dfd, bs, found orelse snd(result)))
       end;
```

We first consider firing an idle bubble on a given rule `r`. The function `enabled` checks whether the enabling condition of the rule passed to it is satisfied. The code for `enabled` will be presented next. Function `Bubble.BInFlows` returns the inflows of a bubble as a list of their names and types, while `Bubble.BName` just returns the name of a bubble. If the bubble is enabled on the given rule, then we use `firerule` (presented after `enabled`) to fire the bubble on that rule. Otherwise, we recursively call `onebubble` to check the next rule.

⟨*firing an idle bubble*⟩≡
```
    if enabled(f, r, Bubble.BInFlows(bub)) then
      let val d = firerule({Flows=f, Bubbles=b}, Bubble.BName(bub), r)
        in (Set.Union(meanaux(d, BubbleSet.BList(bubbles(d)), false),
                      fst(onebubble({Flows=f, Bubbles=b}, bub, rs))),
            true)
    end else (fst(onebubble({Flows=f, Bubbles=b}, bub, rs)), false)
```

The arguments to function `enabled` are the set of flows, the rule to fire on, and the set of inflows of the bubble in question, and so its type is `FlowSet.FSet * Rule * (string * string) list -> bool`. As nontrivial enabling conditions have two parts, a `flowenabled list` and a `preexpr`, we use function `checkinflows` to check that the inflows satisfy the `flowenabled list` portion, and function `preeval` to check that the `preexpr` portion is satisfied. Function `extractpre` gets the `preexpr` portion of the enabling condition — not the rule's pre-condition. The pre-condition will be checked when the bubble is actually fired. Function `getEnv` constructs the environment in which the `preexpr` part of the enabling condition is evaluated.

⟨*function enabled*⟩≡
```
   fun enabled(f, (T, _, _), _) = true
     | enabled(f, r, inf) =
         checkinflows(f, r, inf) andalso
         preeval(extractpre(r), getEnv(f, inf, r));
```

Function `checkinflows` merely checks that any flows tagged with `plus` in the `flowlist` contain tokens, and than any tagged with `minus` don't. The function `FlowSet.isToken` checks whether a flow specified by its name and type is carrying a token. Function `Bubble.findType` is used to find the type of one of the inflows of some bubble.

⟨*function checkinflows*⟩≡
```
fun checkinflows(_, (T, _, _), _) = true
  | checkinflows(_, (flowlist([], _), _, _), _) = true
  | checkinflows(f, (flowlist(plus(s)::fs, p), pre, post), inf) =
      FlowSet.isToken(f, s, Bubble.findType(inf, s))
      andalso checkinflows(f, (flowlist(fs, p), pre, post), inf)
  | checkinflows(f, (flowlist(minus(s)::fs, p), pre, post), inf) =
      not (FlowSet.isToken(f, s, Bubble.findType(inf, s)))
      andalso checkinflows(f, (flowlist(fs, p), pre, post), inf);
```

The function `getEnv` constructs an environment from the list of inflows and the tokens on those flows — the environment that the enabling, pre-, and post-conditions will be evaluated in. Thus, the type of `getEnv` is `FlowSet.FSet * (string * string) list * Rule -> Env.E`. Only flows tagged with `plus` in the enabling condition contribute to the final environment. Function `FlowSet.currToken` returns the token at the head of the queue associated with a flow.

⟨*function getEnv*⟩≡
```
fun getEnv(f, inf, (T, _, _)) = Env.empty
  | getEnv(f, inf, (flowlist([], _), _, _)) = Env.empty
  | getEnv(f, inf, (flowlist(minus(s)::fs, p), pre, post)) =
      getEnv(f, inf, (flowlist(fs, p), pre, post))
  | getEnv(f, inf, (flowlist(plus(s)::fs, p), pre, post)) =
      Env.extend( getEnv(f, inf, (flowlist(fs, p), pre, post)),
      s,
      FlowSet.currToken(f, s, Bubble.findType(inf, s)));
```

Assuming that the enabling condition is satisfied (`enabled` returns `true`), `onebubble` uses function `firerule` to fire the bubble. The `string` argument to `firerule` is the name of the bubble, and it returns the configuration resulting from the firing.

⟨*firerule declaration*⟩≡
```
val firerule: D * string * Rule -> D
```

Function `firerule` uses `BubbleSet.find` to find the bubble being fired by name. Both idle and working bubbles may be fired using `firerule`. As we are currently discussing firing an idle bubble, we discuss that case now, and defer the rest of `firerule` until the discussion of firing working bubbles.

⟨*function firerule*⟩≡
```
fun firerule({Flows=f, Bubbles=b}, name, r) =
  let val cb = BubbleSet.find(b, name) in
  if Bubble.isIdle(cb) then
    ⟨idle case⟩
  else
    ⟨working case⟩
end;
```

In the idle case, function `BubbleSet.replace` produces a new `BubbleSet` with the formerly idle bubble now working, and `consume` produces a new `FlowSet` with the appropriate tokens removed.

⟨*idle case*⟩≡
```
 {Bubbles = BubbleSet.replace(b, name, makeWorking(cb, r, f)),
  Flows = consume(f, r, Bubble.BInFlows(cb))}
```

Thus, a new DFD configuration is produced as follows:

1. Produce a new `FlowSet` that reflects what the bubble being fired has read from its inflows. The type of operation `consume` is
   `FlowSet.FSet * Rule * (string * string) list -> FlowSet.FSet`. The function `consume` uses the `removeToken` operation on flow sets to model consuming tokens from the consumable inflows. Function `removeToken` has no effect on persistent flows. Note that only the flows mentioned in the `flowenabled list` portion of the enabling condition are considered.

   ⟨*function consume*⟩≡
   ```
     fun consume(f, (T, _, _), inf) = f
       | consume(f, (flowlist([], p), _, _), inf) = f
       | consume(f, (flowlist(minus(s)::fs, p), pre, post), inf) =
           consume(f, (flowlist(fs, p), pre, post), inf)
       | consume(f, (flowlist(plus(s)::fs, p), pre, post), inf) =
           consume(FlowSet.removeToken(f, s, Bubble.findType(inf, s)),
                   (flowlist(fs, p), pre, post), inf);
   ```

2. Produce a new `BubbleSet` with the bubble being fired now working. Function `makeWorking` calls `getEnv`, which constructs the environment needed by the bubble to evaluate its pre- and post-conditions.

   ⟨*function makeWorking*⟩≡
   ```
     fun makeWorking(b, r, f) =
        Bubble.makeWorking(b, r, getEnv(f, Bubble.BInFlows(b), r));
   ```

   Function `Bubble.makeWorking` returns a new bubble in the working state, and with the environment constructed by `getEnv` stored in its `Envmt` field. Note that `makeWorking` exists only to call `Bubble.makeWorking` with the appropriate arguments.

Thus, after an idle bubble has fired, it is working. To see how to fire a working bubble, we return to the appropriate case in function `onebubble`. As only one DFD configuration can result from firing a working bubble, `onebubble` simply calls `meanaux` with that configuration.

⟨*firing a working bubble*⟩≡
```
  let val d=firerule({Flows=f, Bubbles=b},Bubble.BName(bub), r)
    in (meanaux(d, BubbleSet.BList(bubbles(d)), false), true) end
```

The interesting part, then, is what `firerule` does when called with a working bubble. Again, we call `BubbleSet.replace` to return a new `BubbleSet` with the `Persistency` of the bubble being fired changed, and `produce` plays the role of `consume` in that it returns a new `FlowSet` with the appropriate changes. Note that the argument of type `Rule` to `firerule` is ignored in this case, as the rule that the bubble became enabled on was stored in the bubble's `CurRule` field by `Bubble.makeWorking` — when the bubble changed state from idle to working.

⟨*working case*⟩≡
```
   if preeval(prec(Bubble.BCurrentRule(cb)), Bubble.BEnv(cb))
     then {Bubbles = BubbleSet.replace(b, name, Bubble.makeIdle(cb)),
           Flows = produce(f, Bubble.BCurrentRule(cb), cb)}
     else raise Precondition_not_satisfied
```

A new DFD configuration is produced as follows:

1. Check the pre-condition. This is done with the call to `preeval`, using the environment that was built by `getEnv` and stored in the bubble's `BEnv` field when the bubble changed state from idle to working. As `preeval` simply checks if a propositional logic assertion is satisfied in the given environment, the code is simple, and so omitted. If the pre-condition

is not satisfied, the exception `Precondition_not_satisfied` is raised. Thus, we see that a bubble expects its pre-condition to be true — for a diagram to finish execution, all pre-conditions must be true when they are evaluated. It is the responsibility of the set of bubbles whose outflows are the inflows to the bubble in question to ensure that the pre-condition of a rule is satisfied whenever its enabling condition is.

2. Produce a new configuration in which the bubble is idle and the output of the bubble is reflected on its outflows. The call to `Bubble.makeIdle` in `firerule` handles the bubble, and the outflows are handled by the call to `produce`, which has type `FlowSet.FSet * Rule * Bubble.B -> FlowSet.FSet`. Additionally, `produce` calls function `posteval`, which is a weak check of the satisfiability of the post-condition. If this check fails, then the exception `Postcondition_not_satisfied` is raised. Usually, this is a sign of a poorly written or incorrect post-condition — we expect that post-conditions are always satisfiable.

   ⟨*function produce*⟩≡
   ```
   fun produce(f, (e, pre, post), b) =
      if posteval(post, Bubble.BEnv(b)) then
        prodaux(f, producelist(post, Bubble.BEnv(b)), b)
        else raise Postcondition_not_satisfied;
   ```

   Function `producelist` walks the post-condition, producing a list of outflow name and token pairs representing the bubble's output. It will be presented after the code for function `prodaux`, which takes the output from `producelist` and uses function `FlowSet.putToken` to produce a new `FlowSet`. Hence, its type is `FlowSet.FSet * (string * Token.T) list * Bubble.B -> FlowSet.FSet`. Function `FlowSet.putToken` enques tokens on consumable flows, and creates a new, one element queue for persistent flows.

   ⟨*function prodaux*⟩≡
   ```
   fun prodaux(f, [], b) = f
     | prodaux(f, (p, t)::ps, b) =
         prodaux(FlowSet.putToken(f, p,
                             Bubble.findType(Bubble.BOutFlows(b), p), t),
                 ps, b);
   ```

   Function `producelist`, of type `postexpr * Env.E -> (string * Token.T) list`, produces the input for function `prodaux` from the post-condition. Function `tokeneval` evaluates `tokenexpr`s into tokens, using the environment passed to `producelist`. Parentheses and square brackets are SML's tuple and list constructors, respectively. Thus, the expression `[(p, tokeneval(i, bv))]` produces a one element list, where that element is an outflow name and token pair. In the alternative for `postand`, the SML operator `@` is used to append two lists, and in the last two alternatives, `[]` is SML's empty list constructor. Note that `producelist` uses `posteval` to ensure that subexpressions of the post-condition that may be false are not used to produce tokens. For example, a post-condition such as

   ```
   postor(postand(postp false, postassign(f1, primi 3)),
          postassign(f1, primi 2))
   ```

   causes `producelist` to send only the value 2 to outflow `f1`.

⟨*function producelist*⟩≡
```
  fun producelist(postassign(p, i), bv) = [(p, tokeneval(i, bv))]
    | producelist(postand(b1, b2), bv) = producelist(b1, bv)
                                         @ producelist(b2, bv)
    | producelist(postor(b1, b2), bv) =
        if posteval(b1, bv) then producelist(b1, bv)
        else producelist(b2, bv)
    | producelist(postimplies(b1, b2), bv) =
        if posteval(b1, bv) then producelist(b2, bv)
        else []
    | producelist(_, _) = [];
```

Thus, function `Meaning` produces the set of all possible final configurations of a DFD. However, the user may wish to make smaller, more controlled experiments with a DFD. Function `Run` does exactly this, as the user chooses what rule to fire at every step. This is more practical than `Meaning`, because it avoids the combinatorial explosion inherent in constructing the set of all possible final configurations.

⟨*Run declaration*⟩≡
```
  val Run: D -> D
```

The operation `RandRun` simulates an actual execution of the diagram by nondeterministically choosing which rule to fire at each step. The process halts (if at all) when no more rules may fire. Thus, `RandRun` automates the kind of experiment possible with `Run`.

⟨*RandRun declaration*⟩≡
```
  val RandRun: D -> D
```

The operation `runaux` allows users to build their own functions for picking which rule to fire. The second argument of `runaux` is a function that picks which rule of a list of bubble names and rules to fire.

⟨*runaux declaration*⟩≡
```
  val runaux: D * ((string * Rule) list -> int) -> D
```

Thus, we have formally defined the meaning of, and provided tools for experimenting with RT-SPECS specifications

4 Conclusion

4.1 Contributions

We wish to highlight three contributions of this research. Two pertain directly to the specification of DFDs, and the third bears on the field of semantics in general.

The first contribution to the specification of DFDs is the previously mentioned simplification of the enabling conditions — namely that it does not make sense to allow arbitrary logical connectives in the `flowenabled list` portion (the part consisting of the list of inflows of the bubble tagged with `plus` or `minus`). This realization occurred while automating evaluation of enabling conditions, and so is a practical result of our effort to formalize the semantics of RT-SPECS. Consider, for example, an enabling condition of the form `or(plus(f1), plus(f2))` (manufacturing some syntax, as we will shortly show why we do not allow this form). This enabling condition is true when a token is present on either flow, or on both. Thus, when the rest of the rule is written, it is unknown which flow actually has a token on it, and so no other part of the rule may refer to the value of a token consumed from either flow. Thus, the information carried on the flows is lost. As an extreme example, when the enabling condition is just "true", no flows may be referenced in the rule. So, the logical connective for the elements of the `flowenabled list` must be "and". A specifier can get the effect of an "or" here by using two rules: one with `flowenabled list plus(f1)`, and the other with `plus(f2)`, with the rest of the rule bodies able to refer to one flow or the other. Hence, there is no need to use "or" in this part of the enabling condition, and a list structure suffices, as there is only one way that its elements can be related.

The other contribution to the specification of DFDs is an insight into the nature of firing rules. By adding the notion of firing a bubble, we have raised the questions: "Can a bubble fire concurrently with itself? Can a bubble fire on two distinct rules simultaneously?" For now, the answer is no — the semantics we have given prohibits this behavior. When a bubble in the working state fires, it must fire on the rule it was using when the transition to the working state was made, and must produce its output and change state to idle. Work on refinement notions for RT-SPECS [10] has shown that if a bubble is refined to a sub-diagram with several bubbles the sub-diagram can exhibit behaviors that look like reading more than one input before producing output. We have investigated formal semantics for bubbles firing concurrently with themselves in a forthcoming work [8], and the application of this work to RT-SPECS is straightforward.

Finally, our use of SML demonstrates an unusual and exciting way of doing semantics. The advantages for the semanticist are twofold:

1. formal semantics done in SML may be automatically type checked, giving us greater precision and more confidence in the semantics,
2. SML provides a language with a clearly defined (and enforced) syntax and semantics (provided operationally by the SML compiler) for communicating with other semanticists

Advantages for specifiers and implementors are given at the end of Section 4.

4.2 Directions for further research

We have given the abstract syntax and semantics for a useful subset of the RT-SPECS model, but more remains to be done. Some of what follows arises from traditional DFDs, but many of the problems are unique to the formalized model.

4.2.1 Research arising from traditional DFDs

Traditional DFDs include a specialized form of bubble known as a *terminator*. Terminators are always either sources or sinks of the diagram — they represent interaction with the world outside the software system. Thus, they are easily modeled in our semantics by bubbles with no firing rules. For sources, tokens may be placed on the outflows using `putToken`, and for sinks, tokens may be observed and removed with `currToken` and `removeToken`.

An interesting possibility stemming from our research is that of using our formal semantics with traditional CASE tools. In particular, Teamwork^{tm} has already been modified to work with RT-SPECS specifications. We would like to parse Teamwork output to the SML input to our semantics.

The labeling of flows is another issue — is it reasonable to require that the combination "name: type" be unique? Specifiers may have a use for multiple flows with the same name and type. Also, the semantics we have presented requires only that all flows incident on the same bubble have unique names (as they are referenced by name only in the firing rules). However, it may be more "natural" for imperative language programmers to label data with its name and type, as this is the way they are used to declaring variables. Additionally, this label is more informative, as both the diagram and the SML rendering of an RT-SPECS specification allow the destination of a flow to be determined at a glance. Thus, we label flows by name and type, but are not yet set in this decision.

We would also like to address the issue of *stores*. In the traditional DFD model, stores are passive holders of data, and so are usually implemented as files. The problem with stores is not how to represent them — intuitively, a bubble with a persistent self-flow of type `StoreElement` should be sufficient. Instead, the main difficulty is providing access to the store. Does each bubble that accesses the store require flows to and from it? How cumbersome would this be in practice? How can we handle concurrent access to stores? Is there a good general technique that solves this problem, or must we rely on ad hoc solutions for each store? We have addressed these questions in another work [8] by modeling stores as flows with multiple origin and destination bubbles, and expect that this approach will work for RT-SPECS as well.

4.2.2 Research arising from RT-SPECS

The problems we have uncovered in formalizing DFDs are perhaps even more interesting than those found in traditional DFDs. We have extended a static model to a dynamic one, and so face issues that don't even exist in the traditional DFD world.

For example, we would like to write *invariants* over our RT-SPECS specifications. An invariant is a logical assertion defining allowable states of the diagram, and so only makes sense in terms of a dynamic model. For example, we might like to require that a certain flow never be empty, or that it never contain more than some fixed number of tokens — consider the bounded buffer producer/consumer problem. Since our specifications are now executable, we would now also like to check that the invariant holds at every step in the computation of the diagram. Hence, the invariant would be evaluated over:

- the initial configuration of the diagram
- the configuration resulting from a bubble consuming from its inflows
- the configuration resulting from a bubble producing onto its outflows

as these are the points where change occurs. Extending the idea of invariants, we may eventually want to use temporal logic [12] [15] to describe liveness and safety properties of RT-SPECS specifications.

Invariants are closely related to *expression definitions*, which are named, parameterized assertions. The advantages of using expression definitions are that they allow modularization of assertions, and that they allow recursive assertions — just as named functions allow recursion in programming languages. The problem with expression definitions lies not in implementing them, but in evaluating them, as this evaluation need not terminate. Consider, for example, a simple definition such as:

```
define loop() such that loop = loop()
```

So, we lose the guarantee that `preexpr`s and `postexpr`s always evaluate to true or false, as they may now fail to halt. Since there is clearly no automated way to check for this condition, users of the semantics would have to be aware of this complication.

Expression definitions are not the largest hurdle remaining in our work. We have yet to deal with allowing logical quantifiers in our firing rules — raising them from propositional logic assertions to full first order predicate calculus ones. Quantifiers are convenient for writing assertions over the elements of a sequence, and vital for assertions over set elements, as the current RT-SPECS model does not have set analogs for the sequence operations **`first`**, **`last`**, **`header`**, and **`trailer`** that allow access to set elements. So, we propose to allow quantification over sets, and as a convenience, sequences and subranges of the integers. This raises the problem of "constructive quantification" — assertions like

$$\forall i [1 \leq i \leq 10 \Rightarrow \texttt{index}(A, i) = i + 1]$$

that not only quantify over a set or sequence, but actually construct one. Thus, the work required for adding quantification is substantial, as we need syntax for the quantifiers themselves and stronger versions of the functions `preeval` and `posteval` to evaluate the new types of expressions. We have studied common quantifier forms and ways of implementing them in the context of an executable specification language for C++, and much of the syntax and evaluation technique developed there applies directly to RT-SPECS.

4.3 Implications of this work

While considerable work remains to be done, the current state of our research already has two important implications. The first is for the use of traditional DFDs as a formal specification technique. By formalizing the model and providing a semantics for it, we have made the rather "warm and fuzzy" Data Flow Diagram into a solid tool for specifying software systems. Thus, we have all the advantages of traditional formal specification — an unambiguous statement of functionality, clean interfacing for team coding projects, and the possibility of proving code correctness. At the same time, our model still allows users to do traditional DFDs — simply by building a diagram with bubbles and flows, but no rules, or even simplified rules that don't capture the entire functionality needed. This informal specification can be directly refined to a formal one — by adding the necessary rules [10].

The second result of our work is that we now can produce executable specifications. Thus, the specification is a prototype of the system. Some of the advantages for the specifier and client are:

1. Validating specifications. The specifier can now test and debug a specification in much the same way that a programmer would validate a program. This pushes validation into the second stage of the software development cycle.
2. Understanding formal specifications. The client, who is likely to have little or no experience with formal methods, now has a way to understand a formal specification. By experimenting with the prototype, the client can discover and report to the specifier erroneous or unexpected results, and missing or incomplete features.

Both these points imply that errors are likely to be discovered earlier than with a traditional software development cycle, resulting in quicker and less expensive production of software.

Thus, it is with high expectations that we approach the next stages of our research. As it stands, our semantics is not ready to handle "industrial strength" specifications — we need quantifiers and the ability to work with a CASE package at the least, but we are confident that we can supply the additional rigor and power needed. The research directions suggested in this paper will make solid progress in that direction.

References

[1] Baker, Albert L. Synthesizing model-oriented and Structured Analysis specifications, Lecture, (1991).

[2] Coleman, David L. Formalized Structured Analysis specifications, Ph.D. dissertation, Iowa State University, Ames, Iowa, 50011, (1991).

[3] Györkös, J., Rozman, I. and Welzar, T. Activation and validation of the system specifications, Microprocessing and Microprogramming 31, (April 1991), North-Holland, 59 - 64.

[4] Hatley, D. and Pirbhai, E. *Strategies for Real-Time System Specification.* Dorset House, New York, 1987.

[5] Hayes, I. *Specification Case Studies.* Prentice-Hall, International Series in Computer Science, 1987.

[6] Jones, Cliff B. *Systematic Software Development Using VDM.* Prentice-Hall, International Series in Computer Science, 1986.

[7] Kung, Chenho. Process interface modeling and consistency checking, The Journal of Systems and Software, 15, 2, (May 1991).

[8] Leavens, Gary T., Wahls, Tim, Baker, Albert L. and Lyle, Kari. A structural operational semantics of firing rules for Structured Analysis style data flow diagrams, Forthcoming, Iowa State University, Ames, Iowa, 50010, (1992).

[9] Lerner, Richard Allen. Specifying Objects of Concurrent Systems, Ph.D. dissertation, Carnegie Mellon University, (1991), Number CMU-CS-91-131.

[10] Lyle, Kari. Refinement in data flow diagrams, Masters thesis, Iowa State University, Ames, Iowa, 50011, (1992).

[11] McDonald, C., and Allison, L. Denotational semantics of a command interpreter and their implementation in Standard ML, The Computer Journal, 32, 5, (October 1989), 422-431.

[12] Manna, Zohar., and Pnueli, Amir. The temporal logic of reactive and concurrent systems, Springer-Verlag, (1992).

[13] Paulson, L. C. *ML for the Working Programmer.* Cambridge University Press, Cambridge, 1991.

[14] Peterson, J. L. Petri nets, ACM Computing Surveys, 9, 3, (September 1977), 221-252.

[15] Pnueli, A. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. *Current Trends in Concurrency: Overviews and Tutorials*, LNCS, Volume 224, de Roever, W. P. Ed., and Rozenberg, G. Ed., Springer-Verlag, 1986, 510-584.

[16] Ramsey, Norman. Literate programming tools need not be complex, to appear in IEEE Transactions on Software Engineering.

[17] Schmidt, David A. *Denotational Semantics — A Methodology for Language Development.* Wm. C. Brown Publishers, Dubuque, Iowa. 1986.

[18] Spivey, J. Michael. *The Z Notation: A Reference Manual.* Prentice-Hall, International Series in Computer Science, 1989.

[19] Spivey, J. An introduction to Z and formal specifications, Software Engineering Journal, (January 1989).

[20] Ward, Paul T. The transformation schema: an extension of the data flow diagram to represent control and timing, IEEE Transactions on Software Engineering, SE-12, 2, (February 1986), 198-210.

[21] Yourdon, Edward. *Modern Structured Analysis.* Yourdon Press computing series. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Tech Report: TR93-27
Submission Date: November 15, 1993