Formalized structured analysis specifications

David L. Coleman

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons, and the Mathematics Commons

Recommended Citation

https://lib.dr.iastate.edu/rtd/9634

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Formalized structured analysis specifications

Coleman, David L., Ph.D.
Iowa State University, 1991
Formalized

structured analysis specifications

by

David L. Coleman

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:  Members of the Committee:
Signature was redacted for privacy.
In Charge of Major Work  Signature was redacted for privacy.
For the Major Department  Signature was redacted for privacy.
For the Graduate College

Iowa State University
Ames, Iowa
1991

Copyright © David L. Coleman, 1991. All rights reserved.
TABLE OF CONTENTS

ACKNOWLEDGEMENTS ................................................. xi

CHAPTER 1. INTRODUCTION ................................. 1
  1.1 Specifications .............................................. 1
  1.2 Two Views of Specifications ............................ 2
  1.3 Formal Specifications .................................... 3
  1.4 Structured Analysis Specifications .................... 5
    1.4.1 History of Structured Analysis .................... 6
    1.4.2 Structured Analysis Today .......................... 7
  1.5 Overview of Dissertation ................................ 10

CHAPTER 2. TRADITIONAL STRUCTURED ANALYSIS ......... 13
  2.1 Overview .................................................. 13
  2.2 Structured Analysis Modeling Tools ................... 13
    2.2.1 Data Flow Diagrams ................................ 14
    2.2.2 Data Dictionary ..................................... 18
    2.2.3 Process Specifications .............................. 20
    2.2.4 Leveled Data Flow Diagrams ....................... 24
    2.2.5 Extensions to SA Specifications ................. 26
  2.3 Ambiguities and Informalities in SA Modeling Tools .... 28
iv

5.2.1 Value Named Flows ........................................ 93
5.2.2 Type Named Flows ........................................ 96
5.2.3 Modified Type Named Flows ............................... 98

5.3 Flow Value Behavior ........................................ 101
5.3.1 Continuous versus Discrete Existence .................... 102
5.3.2 Continuous versus Discrete Change ........................ 104
5.3.3 Continuous versus Discrete Domain ....................... 106
5.3.4 Interpreting Flow Value Behaviors ....................... 107

5.4 Flow Abstraction ........................................... 110
5.4.1 Vergence Flow Abstraction ................................ 113
5.4.2 Decomposition Flow Abstraction ......................... 157
5.4.3 Other Composition Expression Operators ................ 160

5.5 Flow Balancing Summarized ................................ 162

CHAPTER 6. SYNTHESIS OF STRUCTURED ANALYSIS AND
OBJECT-ORIENTED SPECIFICATIONS ......................... 169
6.1 Overview .................................................. 169
6.2 Traditional SA Data Structuring ............................ 171
6.3 Comparing Composition Expressions and SPECS ............ 173
   6.3.1 Structured Types .................................... 173
   6.3.2 Primitive Types ..................................... 177
6.4 Using SPECS ADTs .......................................... 180
6.5 Merging Objected-Oriented Specifications and SA .......... 182

CHAPTER 7. SPECIFYING PROCESS BUBBLES .................. 185
7.1 Overview .................................................. 185
LIST OF TABLES

Table 3.1: BNF Meta-symbols ........................................ 49
Table 3.2: ADT Operation Types ................................. 56
Table 4.1: DDE Composition Expression Syntax ............. 84
Table 5.1: Flow Value Change versus Flow Value Existence .... 107
Table 5.2: Layered Bundling Timing Example .......... 152
Table 6.1: Syntactic Definition of integer .................... 179
Table 6.2: Integer Subrange Definition ..................... 179
Table 7.1: Kung's Consistency Proof ....................... 205
Table 7.2: Consistency Proof Fails ........................... 206
Table 7.3: Producer Consistency Proof .................... 212
Table 8.1: Interface Behavior Syntax ...................... 227
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>DFD of an Order Processing System</td>
<td>8</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Procedural Annotation</td>
<td>15</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Diverging, Converging, and Dialogue Flows</td>
<td>17</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Basic Decision Table Format</td>
<td>22</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>Example Decision Table</td>
<td>22</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>Example Decision Tree</td>
<td>23</td>
</tr>
<tr>
<td>Figure 2.6</td>
<td>A Hierarchy of DFDs</td>
<td>25</td>
</tr>
<tr>
<td>Figure 2.7</td>
<td>Example Entity Relationship Diagram</td>
<td>27</td>
</tr>
<tr>
<td>Figure 2.8</td>
<td>Control Modeling in SA Specifications</td>
<td>28</td>
</tr>
<tr>
<td>Figure 2.9</td>
<td>Example State Transition Diagram</td>
<td>29</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>General Form of a SPECS ADT</td>
<td>37</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>A Recursive Expression Definition</td>
<td>47</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Example SA Specification</td>
<td>61</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>DFD Diagraming Techniques</td>
<td>64</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Converging and Diverging Flows</td>
<td>67</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Excessive Flow Restriction</td>
<td>75</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Example P-Spec</td>
<td>77</td>
</tr>
</tbody>
</table>
Figure 5.25: Timing of a Composite Copy/Split/Combine/Merge Vergence 142
Figure 5.26: Timing of a Composite Select/Copy/Alternate Vergence . . . 143
Figure 5.27: Timing of a Composite Copy/Split Vergence . . . . . . . . 143
Figure 5.28: An Ambiguous Combine Converge . . . . . . . . . . . . . . 145
Figure 5.29: A Non–simple Convergence . . . . . . . . . . . . . . . . . . . 147
Figure 5.30: Equivalent Simple Convergences . . . . . . . . . . . . . . . 147
Figure 5.31: Equivalent Simple Vergences . . . . . . . . . . . . . . . . . 148
Figure 5.32: Timing of Figure 5.30 . . . . . . . . . . . . . . . . . . . . . . 149
Figure 5.33: Timing of Figure 5.31 . . . . . . . . . . . . . . . . . . . . . . 149
Figure 5.34: Layered Bundling Example . . . . . . . . . . . . . . . . . . . 151
Figure 5.35: Layered Bundling Equivalent Simple Vergences . . . . . . 152
Figure 5.36: Strict Consumable Vergence Balancing Algorithm . . . . . 154
Figure 5.37: Equivalent Simple Strict Vergences . . . . . . . . . . . . . 156
Figure 5.38: Other Equivalent Simple Strict Vergences . . . . . . . . . 156
Figure 5.39: Timing for Growing Queue Example . . . . . . . . . . . . . 157
Figure 5.40: Timing of Optional Convergence . . . . . . . . . . . . . . 161
Figure 5.41: Timing of Optional Divergence . . . . . . . . . . . . . . . . 162
Figure 5.42: Disjunctive Normal Form Algorithm . . . . . . . . . . . . . 167
Figure 6.1: Dual Composition Expression Operator Usage . . . . . . . . 171
Figure 6.2: The DFD/ADT Boundary . . . . . . . . . . . . . . . . . . . . . 183
Figure 7.1: The producer–consumer Example . . . . . . . . . . . . . . . 198
Figure 7.2: Corrected producer–consumer Example . . . . . . . . . . . 199
Figure 7.3: A Decomposition of the Producer Process . . . . . . . . . . 202
Figure 7.4: A Corrected Decomposition of the Producer Process . . . . . 208
Figure 8.1: A Storeless Producer-Consumer DFD . . . . . . . . . . 221
Figure 8.2: A DFD Decomposition of a Store . . . . . . . . . . . . . . . 224
ACKNOWLEDGEMENTS

I am deeply indebted to my thesis advisor and friend, Dr. Albert L. Baker. Al's encouragement and direction kept me on track during many of the discouraging times. He willingly read numerous versions of the thesis, would meet with me on weekends, and arranged his La Crosse visits to allow time for us to work together. He has been both patient and demanding in all the right measures. Al has gone beyond the role of a mere advisor and I count myself fortunate to be working with him. I look forward to continuing our professional relationship and friendship in many future endeavors. I would also like to thank Dr. Gary Leavens and John Rose for their useful suggestions for improvement and clarification. John was especially helpful in his willingness to read earlier versions of this work.

On a different note, I would like to thank my wife Kathleen Coleman for her love, patience, encouragement, and sacrifices throughout my years at graduate school. Kathleen also spent a large amount of time proof reading the thesis, especially during the final hours of preparation. I would also like to thank my parents Lawrence and Helen Coleman for their love and encouragement.

Finally, I would like to thank Waqar Haque, Milt Wikstrom, and the rest of the Friday Afternoon Club (FAC) for time well spent.

This research was supported in part by Rockwell International.
CHAPTER 1. INTRODUCTION

1.1 Specifications

Specifications define systems. Thus a specification distinguishes and clarifies the essential meaning of a system. The definition of a system can be stated either casually or formally. This is a dissertation on making a popular informal specification technique a more rigorous tool for formally defining systems.

A specification is the first document produced on a project intended to be used by the technicians on the development team. Specifications are derived by analysis of the client’s requirements. A specification communicates the client’s requirements to the system developers. The specification defines the functionality of the system—exactly what the system is to do. In comparison, the design defines the implementation of a system. A specification can be viewed as a contract defining the requirements which the developers must satisfy.

Specifications should be unambiguous, complete, and non-contradictory. To further distinguish specifications from design, we should be able to write specifications that are independent of:

- a particular implementation platform.
- a particular implementation language or paradigm; i.e., imperative, functional,
logic, object-oriented, etc.

- a particular system architecture; i.e., sequential, concurrent, distributed, etc.
- particular algorithms.

Specifications are not required to be independent of these design features, but should be free to exclude them. For example, if the client requires the implementation to be based on a sequential machine architecture, then we may tailor our specification in a sequential manner. In the ideal situation, these design decisions are made after the specification is developed. In these cases, we may be able to provide heuristics, which given a specification and a particular implementation strategy, will help generate a design for the product.

1.2 Two Views of Specifications

There are two de facto camps in the Software Engineering community—advocates of formal specifications, and users of informal specification techniques. The term “formal specification” is not used consistently in the professional community. The meaning of “formal specification”, which distinguishes the two camps, is that a formal specification is a mathematically precise definition of software functionality.

The benefits of formal specifications are clear [8, 10, 30, 37, 55]. Formal specifications:

1. may be mathematically analysed, thus providing valuable information about the structure and complexity of proposed systems;

2. may be automatically processed, thus tools can be built to assist in their development;
3. may be executable, thus providing a prototype of the system;

4. may allow for automated consistency checking and verification of implementa-
tions; and

5. may provide semi-automated library retrieval for support of software reuse.

Some arguments against the use of formal specifications have been refuted [26].
Opponents of formal specifications give two main reasons why formal specifications
cannot be used:

1. Specifications should serve as a contract between software developers and soft-
ware clients (users), but formal specifications cannot be understood by the
typical software client.

2. Specifications should serve as a means of communication between individuals
within the software development company, but formal specifications are based
on mathematical constructs which are difficult to learn.

Software systems are inherently complex. Any precise definition of such a system
must rely on a formal, i.e. mathematical, definition of functionality. The former point
has long been an argument for the use of Structured Analysis (SA) specifications. SA
specifications are an informal graphical technique for defining a system in terms of
the flow of information between system components.

1.3 Formal Specifications

A formal specification mathematically defines system functionality. Model based
formal specifications, such as Z [28], VDM [33], and SPECS [6], define a system by
mathematically modeling the state of the system (the objects) and the operations that transform the system from one state to another. For example, a specification in SPECS models a system as an Abstract Data Type (ADT). The abstract domain of the ADT defines the set of states the system may be in. Thus an instance of the ADT defines a single state. The abstract operations define how the system can be transformed from one state to another. SPECS uses mathematical objects to define the domain, and first order predicate calculus assertions to axiomatically define the operations. A complete definition of SPECS is provided in Chapter 3.

Existing formal specification techniques do not lend themselves well to the development of distributed/concurrent systems. Those techniques that do claim to provide distributed/concurrent specifications typically address issues of low level issues, e.g., deadlock and synchronization [14, 39, 41, 52, 63]. While these issues are important, they are design issues. Specifications for distributed/concurrent systems should be able to represent the potential for parallelism, independent of these low level concerns.

Specification techniques do exist for specifying the timing of distributed/concurrent systems. A popular technique for specifying the timing of distributed or embedded systems is a Petri net [53, 63]. A Petri net is a bipartite directed graph. The nodes of the graph are divided into two types of nodes called places and transitions. Each edge in the graph connects a place with a transition. The edge may be directed in either direction. A Petri net execution is based on the placement of tokens in the place nodes (called a marking) and the firing of transition nodes. The ability of a transition to fire is predicated on the placement of tokens in places having edges directed to the transition (input places). When a transition fires, it removes
tokens from input places and places tokens in output places. An output place of one transition may be an input place for another transition. Thus firing one transition may enable another.

Various modifications to Petri net notation exist for describing firing rules that require only some input places to have tokens and only place tokens in some output places. The execution of a Petri net is based on an initial marking of the places with tokens. A single transition is non-deterministically chosen from the set of all transitions able to fire and that transition is fired. This process is repeated as long as possible, thus describing a potentially infinite sequence of markings. Each transition firing describes a mapping from one marking to the next in the sequence. The sequence of marking defines an execution of the Petri net. Petri nets can describe parallel executions. If a reordering of two adjacent firings have the same result, then these two transitions may be fired in parallel.

Petri nets can be mathematically analyzed for deadlock liveliness, etc. [48]. However, Petri nets lack a representation for functionality. Unlike data flow programming languages, the tokens in Petri nets do not have values.

Data flow programming languages also provide for parallel execution [20]. The data flow model provided by SA specifications can also be interpreted in parallel. However, as stated previously, this should not dictate a distributed or concurrent implementation, but rather provide the potential for a parallel implementation.

1.4 Structured Analysis Specifications

SA specifications are based on a graphical syntax with little underlying formal structure. However, they do provide a graphical overview of system functionality and
a graphical decomposition of this functionality.

Practitioners claim, that with little training, software clients can read and understand SA specifications. Thus it is claimed that SA specifications facilitate user comprehension of the software contract. However, SA specifications do not have rigorous semantics. What the client “understands” and what the software developer “understands” may not be the same. This problem compounds itself when members of the same development team have different “understandings” of a particular SA specification. Therefore, SA specifications do not provide the desired rigorous definition of systems.

Formalized SA specifications, as defined by this thesis, are not substantially or graphically different from informal SA specifications. Thus any claim that clients will have more difficulty understanding them is minimal. One may question whether clients ever really understood them in the first place. Formalized SA specifications do provide a basis for formal communication among members of the development team.

1.4.1 History of Structured Analysis

The SA methodology began as an out-growth of the structured programming methodology, and structured design methodologies developed during the 1970's. De Marco first introduced SA specifications for the definition of systems in the late 1970's [18]. De Marco combined the graphical aids used by Constantine [65] for transform analysis with the data design techniques of Jackson [31]. Gane and Sarson [22] developed a similar methodology called Structured Systems Analysis with some influence from De Marco.

More recent developments include extensions to traditional SA by Ward and
Mellor [60, 61, 62, 59], and by Hatley and Pirbhai [27], for modeling real-time systems. We will briefly discuss these extensions in Chapter 2. We question whether these extensions are necessary for development of a real-time specification. In Chapter 7 we argue that these extensions are actually a design methodology.

Kung provides an extension to SA that while deficient, leads us to conclude that the extensions for real-time system specifications are superfluous [36]. In Chapter 7 we present Kung's extension, demonstrate its deficiency, and provide a solution that obviates the need for the real-time extensions.

1.4.2 Structured Analysis Today

SA is a widely used method of informal software specification [18, 22, 60, 61, 62, 66]. Its popularity derives mainly from the graphical representation of software system specifications based on information flow. These graphical representations are hierarchical in nature so that a component at one level in the diagram can have more detail provided at the next level. These pictures of information flow are called data flow diagrams (DFDs). The components of a DFD include information flows, process bubbles, terminators, and information stores.\(^1\) Figure 1.1 is an example DFD of the top level view of an order processing system. Circles represent process bubbles, open-end boxes represent stores, closed-end boxes represent terminators, and directed edges represent information flows. In Chapter 2 we present the traditional definition of an SA specification and provide an interpretation of what an SA specification might mean. These interpretations are based on our interpretations of the current

\(^1\)The use of the term process here does not indicate a true distributed/concurrent process, but rather just the loose terminology used by traditional SA.
Schach refers to SA as a semi-formal specification technique [53]. SA specifications are more formal than natural language text because they require a particular graphical syntax, e.g., an information flow cannot connect two stores, and the construction of an SA specification follows a (loosely) defined methodology. They are informal in the sense that there are components of an SA specification which are specified in natural language, that there are no well-defined semantics with which to interpret components of an SA specification, and that there are no operational se-
mantics for interpreting an SA specification as a whole. Simply put, SA specifications are not executionally interpreted.

Modern Structured Analysis (SA), as described by Yourdon [66], uses several modeling tools (components) to develop specifications. These modeling tools include Data Flow Diagrams (DFDs), Data Dictionaries (DDs), Process Specifications (P-Specs), Entity Relationship Diagrams (ERDs), and State Transition Diagrams (STDs). Computer Aided Software Engineering (CASE) tools like Teamwork also include other modeling tools like State Event Matrices (SEMs), Decision Tables (DTs), and Process Activation Tables (PATs) [29]. Most of these tools are graphical in nature and have defined syntactic structures. (The particular structure varies by author.) However, to understand a specification of a system it is important to understand the semantics of the individual modeling tools and how they fit together as a whole.\(^2\)

The meanings of SA specifications are usually presented by example. Thus SA specifications are informal. The result is ambiguous semantics and little automated help in the software development process. Getting from the specification, through the design, to the implementation is still a human activity. The semantic interpretations we present in Chapter 2 are simply our interpretations of what has been presented in the existing literature. In this thesis, we identify and formalize those underlying structures that are represented informally, provide a formal definition of a SA specification, develop formal interpretations for those components of SA specifications that are subject to varying interpretation, and define an operational semantics for animating SA specifications. The resulting formalized SA specifications are mathematically

\(^2\)The Teamwork environment provides some syntactic checking of model components and relationships between components, thus assuring a limited consistency in the development of a system specification.
precise and can be used to specify distributed/concurrent systems.

1.5 Overview of Dissertation

The goal of this thesis is to make SA specifications a rigorous technique for the formal definition of systems. This is accomplished in four (not necessarily sequential) steps:

1. We identify and formalize those components of SA specifications that are represented informally. Informal representations cannot have rigorous semantics because they lack a well-defined syntax. Informally represented SA components are formalized by replacing their existing representations with well-defined syntactic representations based on the syntax of SPECS and finite state machines. The semantics of SPECS and finite state machines can then be applied directly to define the semantics of these SA components.

2. We provide a formal definition of an SA specification. This formal definition is provided by defining a SPECS ADT whose domain defines the set of all valid SA specifications. An SA specification is valid if it has a particular syntactic structure and a semantic interpretation. Thus the domain of valid SA specifications is analogous to the domain of compilable programs. This does not necessarily ensure the semantic interpretation of an SA specification correctly defines a particular system, but rather that it defines some system.

3. We develop formal interpretations for those components of SA specifications that are subject to varying interpretations, including a mathematically sound and practical approach to the formal specification of concurrent processes and
a rigorous characterization of flow value behavior and timing. This is accomplished by using the SPECS formalization of syntactic structure to describe the semantic interpretation of these syntactic structures.

4. We define an operational semantics for SA specifications. Formal semantics for programming languages ensure the meaning of a particular program is independent of a particular machine or compiler. Three standard methods for the specification of programming language semantics are [54]:

(a) operational semantics
(b) denotational semantics
(c) axiomatic semantics

Operational semantics define the meaning of a program in terms of the evaluation of the program on a particular abstract machine. Denotational semantics map a program to a mathematical expression of its value or function. Axiomatic semantics do not explicitly define the meaning of a program, but rather define properties of the language that can be used to infer the meaning of a program.

We define, albeit informally, an operational semantics for SA specifications based on the satisfaction of first order predicate calculus assertions and finite state machine representations. Furthermore, when possible, we incorporate the traditional informal semantics of SA specifications into our formal interpretations. In this way, the acceptance and use of this work by the practicing professional community is facilitated.

In Chapter 2 we present the traditional SA modeling tools and identify specific attributes of SA specifications that lack formal interpretation. In Chapter 3 we
provide a definition of the SPECS language. In Chapter 4 we present an initial formalization of a traditional SA specification. In Chapter 5 we formalize the DFD flow. In particular, we address the issues of naming flows, develop an interpretation for the representation of flow values, and apply this representation of flow values to develop an interpretation of flow abstraction. In Chapter 6 we enhance the traditional representation of data value structure by incorporating objected-based specifications for the representation of data value structure. In Chapter 7 we develop a state-based assertional specification of process bubble behavior. In Chapter 8 we present issues that need to be resolved for the formalization of stores and terminators in DFDs. A preliminary formalization of both components is provided. In Chapter 9 we provide operational semantics for an SA specification. In Chapter 10 we summarize the results of this thesis, identify future research issues that need to be addressed, and discuss the benefits and applications of the results of our research.
2.1 Overview

In this chapter we present the traditional SA modeling tools, and identify specific attributes of SA specifications that lack formal interpretation. As we present each specific modeling tool, we informally characterize the semantics of each tool. These informal semantics are based on our interpretation of the traditional literature. We do not claim that our interpretations are the same as those intended by the original authors. What follows is simply our best effort to characterize the common, albeit not well-defined, understanding of what an SA specification represents.

In Section 2.2 we present the three main modeling tools of traditional SA. For each modeling tool we provide the traditional informal meaning of tool components and describe the formal syntactic rules that must be followed during construction. We also mention some of the extensions to the set of SA modeling tools and their roles in SA specifications. In Section 2.3 we identify and discuss those traditional modeling tool components that lack formal interpretation.

2.2 Structured Analysis Modeling Tools

The three main modeling tools used in traditional SA are:
1. the Data Flow Diagram (DFD)

2. the Data Dictionary (DD)

3. the Process Specification (P-Spec)

Extensions to the set of SA modeling tools include:

- the Entity Relationship Diagram (ERD)
- the Control Flow and Control Process (C-Spec)
- the State Transition Diagram (STD)

In the following sections we provide traditional definitions of each of these tools.

2.2.1 Data Flow Diagrams

The DFD represents the main modeling tool for an SA specification. A DFD represents the flow of data, not control, in a system. It portrays the situation from the point of view of the data the system acts upon, not from the point of view of those who act on the data. The DFD identifies what functions the system must perform, what the interactions are between functions, where the system gets its information, and where it delivers its results. Figure 1.1 provides an example of a DFD.

The four basic components of a DFD are:

1. Bubbles, represented by circles

2. Flows, represented by directed edges

3. Stores, represented by parallel lines
4. Terminators, represented by rectangular boxes

Each component is identified by its label.

Bubbles represent the various functions that the system carries out. Bubbles are often referred to as processes or data transformations. We object to the general application of the term process to bubbles because of the connotations of the term process in distributed/concurrent systems. Thus we will use the term process bubble or just bubble when referring to the circles in DFDs.

Bubbles transform inputs into outputs. Each bubble is required to have a descriptive name that gives the user a general idea of what it does. A bubble can normally be named in terms of its inputs and outputs, for example, in Figure 1.1, bubble Process.Payment has an input flow labeled Payment.

DFD bubbles can be annotated to represent "some procedural information." De Marco uses * to represent conjunction of flows and © to represent disjunction of flows [18]. For example, in Figure 2.1 the bubble Evaluate_Blackjack_Hand requires both its inputs and produces only one of its outputs.

Flows define the interfaces among the other components of a DFD. Flows are also called data flows to distinguish them from control flows used in real-time extensions.
to SA specifications. We refer to them as just flows whenever the meaning is clear. Flows represent the information that the bubbles require as input and produce as output. Flows may only connect bubbles with bubbles, bubbles with terminators, or bubbles with stores. Flows may not connect terminators with terminators, stores with stores, or terminators with stores.

De Marco refers to a flow as "a pipeline through which packets of information of known composition flow" [18]. The direction of movement is indicated by the direction of the flow. A single flow represents a packet of information that always flows together. If two pieces of information do not always appear at the same time, they must be represented by two different flows.

De Marco requires that each flow be given a unique name [18]. The name of the flow should describe not only the data represented by the flow, but what we know about the data as well. For example, PhoneNumber could be the label of a flow that carries phone numbers, but a flow that carries phone numbers that have been validated by the system should be called ValidPhoneNumber. Flows into and out of stores are not required to have names. These flows are implicitly named by the store.

Flows can also be shown as diverging, converging, or with double-directed arrow heads as in Figure 2.2. A diverging flow means that duplicate copies of a packet of data are being sent to different parts of the system or that a complex packet of data is being split into several more elementary data packets or that a flow carries items with different values that are being separated. A converging flow represents several possible sources for a single data packet or it represents several elementary packets of data that are joining to form a complex packet or it represents several different types of values that are combining to share a pipeline. A double-directed flow, called
a dialogue flow, represents a packaging of two packets, a query and response, on a single flow. Each end of a dialogue flow must be named.

Flows do not represent the flow of control as in an imperative control flow graph, nor do they represent bubble activation. Flows do not represent prompts for information. If a flow is difficult to name, De Marco says, it probably shouldn't be a flow [18].

Stores represent collections of data that the system must remember for some period of time. Stores are also referred to as files or databases. A store's name should describe the contents of the store similar to the name of a flow. Whereas the flows represent data in motion, stores represent data at rest. Flows into a store represent additions to the contents of the store. Flows out of a store represent access to information in the store. Access to a store is nondestructive—the store does not change because of an out-going flow. Stores are passive and thus additions and access to a store must be initiated by a bubble.

Terminators represent external entities with which the system communicates. Terminators are sources and sinks for information that are outside the control of the system.

General guidelines for construction of DFDs include [18, 66]:

1. Identify all net input and output data flows. Draw them in around the outside
of your diagram.

2. Choose meaningful names for bubbles, flows, stores, and terminators.

3. Redraw the DFD as many times as necessary for esthetics.

4. Avoid overly complex DFDs.

5. Make sure the DFD is logically consistent.

   (a) Avoid infinite sinks—bubbles with no outputs.

   (b) Avoid spontaneous generation bubbles—bubbles with no inputs.

   (c) Beware of unlabeled flows and unlabeled bubbles.

   (d) Beware of read only or write only stores.

6. Do not show flow of control, bubble activation, or information prompts.

A DFD is a data-driven representation of the system. DFDs are graphical, can
be partitioned, have multiple dimensions, and do not depict control flow. In Section
2.2.4 we describe how to partition DFDs to form a hierarchy of DFDs.

2.2.2 Data Dictionary

The DD provides the definition of data—the details of what information is trans­formed. For example, the flow labeled Payment in Figure 1.1 might be defined as
follows:

\[
\text{Payment} = \text{CustomerName} + \text{CustomerAddress} + \text{OrderNumber} + \text{AmountOfPayment}
\]
Each subordinate of this definition must also be defined in the DD. For example, \( \text{OrderNumber} \) might be defined as follows:

\[
\text{OrderNumber} = \text{StateCode} + \text{CustomerAccountNumber} + \\
\text{SalesmanID} + \text{SequentialCount}
\]

There must be a data dictionary entry (DDE) for every flow, store, and all subordinates used to define them. The bottom of this seemingly infinite pit is provided by the elementary data elements and literal strings. Elementary data elements are self-defining and require no definition. For example, \( \text{CurrentHeight} \), \( \text{DateOfBirth} \), \( \text{FirstName} \), and \( \text{HomePhoneNumber} \) are elementary data elements. Each elementary data element must be entered in the DD and defined by a brief comment describing the meaning of the term.

DDEs are defined using the following notation:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>is composed of</td>
</tr>
<tr>
<td>+</td>
<td>and</td>
</tr>
<tr>
<td>(</td>
<td>optional (may be present or absent)</td>
</tr>
<tr>
<td>{}</td>
<td>iteration</td>
</tr>
<tr>
<td>[]</td>
<td>select one of several alternative choices</td>
</tr>
<tr>
<td>**</td>
<td>comment</td>
</tr>
<tr>
<td>@</td>
<td>identifier (key field) for a store</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>literal string</td>
</tr>
<tr>
<td></td>
<td>separates alternative choices in the [] construct</td>
</tr>
</tbody>
</table>

For example, we might define \( \text{CustomerName} \) using the following collection of DDEs:
CustomerName = CourtesyTitle + FirstName + (MiddleName) + LastName

CourtesyTitle = ["Mr." | "Miss" | "Mrs." | "Ms." | "Dr." | "Professor"]

FirstName = { LegalCharacter }

MiddleName = { LegalCharacter }

LastName = { LegalCharacter }

LegalCharacter = * any legal alphabet character *

The iteration construct can include optional lower and upper bounds which by default are zero and infinity.

A DDE might also include extraneous information about data such as:

• frequency
• volume
• size
• affected users
• peaks and valleys (boundaries)
• security considerations
• priority
• units

2.2.3 Process Specifications

P-Specs (also called mini-specs) provide the specification for each of the DFD bubbles. A P-Spec defines what must be done to transform a bubble's inputs into out-
puts. Various methods are used to produce P-Specs: decision tables, decision trees, structured English, pre/post conditions, flowcharts, Nassi–Shneiderman diagrams, etc. [24]. The first three methods are traditionally used, with structured English being the most popular. No matter which method is chosen, Yourdon remarks that the specification must satisfy two requirements: [66]

1. The process specification must be expressed in a form verifiable by both the user (client) and systems analyst (specifier).

2. The process specification must be expressed in a form that can be effectively communicated to the various audiences involved. These include users, managers, auditors, quality assurance personnel, designers, testers, and others.

We will discuss each of the methods briefly.

Decision tables and decision trees are tabular and graphical techniques for expressing decision logic. A decision table has the basic format given in Figure 2.3. The condition stub vertically lists all conditions to be considered. The condition entries list yes (Y), no (N), or don't care (D) values for each condition in each rule column. No two rule columns should have the same pattern of condition entries. The action stub vertically list all potential actions that may be taken in each case. Action entries are marked (with an X) if the action should be taken when the pattern of condition entries in that rule column are true.

For example, Figure 2.4 provides a decision table specification of the following English narrative.

If a signal is from a device of type A and the system is in Alert mode or if the signal is from a device of type B and the system is in Ready
mode then the signal must receive priority handling. If the signal is from a device of type A, the system is in Ready mode, and the signal was received more than 1 ms ago then the signal must receive priority handling. If the signal is from a device of type B, the system is in Alert mode and the signal was received more than 1 ms ago then the signal must receive priority handling. Under all other conditions the signal must receive normal handling.

<table>
<thead>
<tr>
<th>Determine Signal Handling</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1: device type A?</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>c2: mode = Alert?</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>c3: received ≤ 1 ms ago?</td>
<td>D</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>D</td>
</tr>
<tr>
<td>a1: priority handling</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a2: normal handling</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.4: Example Decision Table

A decision tree is a network-type graph expressing decision logic similar to that expressed by a decision table. Figure 2.5 provides an example decision tree based on the previous English narrative.

Structured English is a method for displaying logic in an outline format. The "structure" of structured English results from the limiting of English to a subset of the
Figure 2.5: Example Decision Tree

full English language with restrictions on the kinds of sentences allowed. Structured English is often known as pseudo-code. Its purpose is to provide a bridge between English and programming languages. Structured English is usually limited to a small set of action verbs and use of simple (noncompound) sentences. These sentences can be combined using traditional structured pseudo-code statements, i.e., IF-THEN, IF-THEN-ELSE, WHILE-DO, etc.

Pre/post conditions are declarative style statements of bubble processing. Although these can be quite mathematically precise, they are usually written in English. Flowcharts and Nassi-Shneiderman diagrams are graphical depictions of pseudo-coded processing logic.
2.2.4 Leveled Data Flow Diagrams

The DFD provides a broad overview of the system, but for complex systems the number of bubbles appearing in a diagram becomes unmanageable. Leveled DFDs allow for abstraction of groups of bubbles at successively higher levels.\(^1\) When a particular DFD becomes too large, we can partition the bubbles into subsystems and represent each subsystem by a single bubble. If these subsystems are too large we can further partition each subsystem, and so on, until we end up with a hierarchy of DFDs as in Figure 2.6. Yourdon offers a general rule of thumb for deciding when to decompose a DFD bubble into another DFD [66].

If we can’t write a reasonable process specification (P-Spec) for a bubble in about one page, then it probably is too complex and should be partitioned into a lower level DFD.

Adler defines an algebraic method for choosing how to decompose a bubble based on information flow analysis [2].

Each bubble that decomposes into a DFD represents an abstraction of the subsystem it decomposes into. To make this hierarchy complete, each bubble in a DFD must be represented by either another DFD at a lower level in the hierarchy or a P-Spec. P-Spec bubbles do not have to occur all at the same level. For example, in Figure 2.6 every bubble that is not represented as decomposing into a DFD would be represented by a P-Spec.

\(^1\)The term \textit{leveled} is traditionally used in SA to describe a hierarchy of DFDs. This is contrary to the typical meanings of leveled, e.g., to place things in the same rank, degree, or plane.
Figure 2.6: A Hierarchy of DFDs
Decomposition of a bubble into a DFD must be balanced. Input flows (in-flows) and output flows (out-flows) of the parent bubble must show up as flows coming from or going off page (off-page flows) in the child DFD. However, this balancing is not always one-to-one. Packets of information may be broken apart, put together, or duplicated in this decomposition. This is similar to convergence and divergence of flows. De Marco calls this parallel decomposition of function and data [18].

De Marco gives the following set of rules for developing a leveled DFD hierarchy:

- Start with a single bubble that represents the entire system surrounded by all the terminators. This defines the interface boundary and is traditionally called the context diagram.
- Label each DFD with the label of the bubble it represents.
- Balance in-flows and out-flows between a parent bubble and a child DFD.
- At any given level only show the stores and flows that are relevant to the bubbles which appear at that level.
- A particular store can only appear in one DFD. All references to that store must also appear in that same DFD.

2.2.5 Extensions to SA Specifications

The extensions to SA modeling tools include ERDs and real-time control modeling [12, 60, 61, 62, 59, 66]. An ERD is a graphical tool for describing relationships between stored data. Typically this means relationships between stores in a DFD. These relationships are not depicted by the DFD. In an ERD, stored data is represented by an object type. Object types represent collections of objects in the real
world whose members play a role in the system and have one or more attributes. Object types are drawn as rectangles. Relationships are drawn as diamond-shaped boxes and represent associations between objects. Figure 2.7 provides a typical ERD.

![ERD Diagram](image)

**Figure 2.7: Example Entity Relationship Diagram**

Control modeling is accomplished by including control bubbles and control flows in DFDs. Control bubbles and control flows are drawn using dashed lines. For example, Figure 2.8 is Yourdon's example of a DFD augmented with control bubbles and control flows [66]. In Figure 2.8 when Bubble1 produces its output, it sends a signal on W to the control bubble. The control bubble then sends an X signal to Bubble2 telling it to produce its output. When Bubble2 produces its output it sends a signal on Y. The control process then sends a Z signal to Bubble3 telling it to begin its work. It's not clear why Bubble2 and Bubble3 cannot self-detect the presence of their own input.

Control bubbles are typically specified by a state transition diagram (STD). Figure 2.9 provides an example STD for the control bubble in Figure 2.8. Each tran-
position is augmented by an input/output pair representing the receiving and sending of signals on control flows. A STD can also be represented in a tabular form.

2.3 Ambiguities and Informalities in SA Modeling Tools

In this section we discuss those components of SA specifications that lack formal interpretation.

2.3.1 Ambiguities in Data Flow Diagrams

DFDs do not provide an operational interpretation of the systems they specify. This is because the components of DFDs are not well-defined and because there is no common definition of "execution" of a DFD. Other formal specification techniques
Figure 2.9: Example State Transition Diagram
allow a black box characterization of a system; DFDs do not. In this section we consider those features of DFD components that are ill-defined.

Flows represent one of the most mysterious components of a DFD. Although each flow label is defined in the DD by a DDE, we still can identify several questions concerning flows. How exactly do flows represent information? Do flow labels represent value names like variables or are they names of type structure? What is the relationship of packets of information to flows? One-to-one? Many-to-one? Can several packets appear on the same flow? At the same time? Can the same packet appear on several different flows? At the same time? If yes, are they copies of the same packet or "pointers" to a single version? What does it mean when a flow diverges or converges? In Chapters 5 and 6 we provide answers to these questions.

Flow direction determines which DFD component produces the data and which DFD component receives the data, but how does this happen? Is data implicitly prompted for or does it arrive of its own free will—indeed, independent of any action on the part of the receiver? Is there a difference in flows between process bubbles versus flows between a process bubble and a store or terminator?

A major problem with DFDs in general is the lack of operational semantics for describing their behavior. Based on our formalizations of DFD components, in Chapter 9 we provide an operational semantics for DFDs that shows exactly how data moves through the system. These operational semantics are completely independent of any particular implementation strategy, rather they just serve to animate the specification.

---

2 A black box characterization of a system defines the expected output of the system in terms of the input without defining how the output is produced.
Bubbles, are they imperative procedures or concurrent processes? Our major concern with bubbles is when do they do their work. We intentionally use the term bubble to describe these processing components of a DFD. The term process has connotations associated with distributed/concurrent processes. Whether these connotations are appropriate for DFD process bubbles is central to the discussions and results of Chapter 7. Does a process bubble represent a sequential imperative procedure or a true distributed/concurrent process? Traditionally, system design from SA specifications created an imperative procedure or function for each process bubble. Even the real-time extensions to SA do not address the potential for implementation as a distributed/concurrent system, although control bubbles begin to have the appearance of true processes [12, 59]. In Chapter 7 we provide a method for formally specifying the dynamic timing (the when) and the functional behavior (the what) of process bubble execution independent of a particular implementation strategy or algorithm (the how).

Stores are often referred to as files. De Marco calls a file a temporary repository of data [18]. The term "file" certainly implies a very strict implementation mentality. We might interpret "temporary" as excluding those files that exist between executions of the implemented software system. However, this is not clear. Figure 1.1, while contrived by the author, does represent a typical DFD and we would certainly expect that the Inventory and Accounts stores will persist when the system is shut down. Note that the system is probably incomplete with respect to the Inventory store. Assuming bubble Process_Order just checks Inventory to see if the item ordered has not been discontinued and bubble Fill_Order decrements the number of items on hand, we have no representation of adding new stock to the inventory on hand.
Yourdon identifies a major concern about stores [66]. Does the store exist because the current manual system has some delay between production and use of the data or is it a convenience to the implementation of the system? We actually feel it is neither. In Chapter 8 we question whether stores are really necessary—can they be modeled by a storeless DFD? This may depend on the interpretation of flows. However, stores may provide for a declarative control representation of concurrent access to the data a store contains.

Other than this major concern, many of the issues for flows and process bubbles can be applied to stores. How do stores represent data? When do they produce their data? When do they receive their data? Do stores process their data like a process bubble or are they in some sense passive?

Terminators are defined only by their label, which is not further defined. The net flows between terminators and process bubbles represent the interface boundary between the implemented system and the system users. By not defining these terminators, we cannot determine if the system will respond appropriately to the external world. In part, real-time extensions have addressed this problem, by providing dynamic time dependent behavior of bubbles, but the solution is not complete because the dynamic time dependent behavior of terminators has not been addressed [12, 59]. Because terminators exist outside the system, it is not always possible to provide a complete functional specification of their behavior. However, it is possible to model the dynamic behavior of their production and consumption of data, excluding the functional value of the data. In some cases, even functional value can be specified. For example, suppose a terminator represented a file system server and the flows into and out of this terminator represented arguments and results of file operations. This
terminator might be part of the specification of a document spelling checker system where multiple document files must be retrieved from the file system. In the case where the terminator is another software (or hardware) system, its own SA specification (assuming it exists) can serve to describe its external behavior and can be incorporated into the new system's model.

The DFD hierarchy provides a multidimensional abstracted view of the system. This abstracted view can be used for system comprehension at different levels of detail. However, we must assure that each layer of abstraction is consistent with its underlying detailed representation. In particular, the decomposition of a parent bubble into a child DFD must be balanced with respect to net in-flows and out-flows. This balancing may also allow abstraction of flows similar to convergence and divergence in a DFD. Interpreting (and balancing) these flow abstractions is more difficult than convergence and divergence in a single level because the association between parent flows and child flows is not graphically shown.

The operational semantics we develop in Chapter 9 are based on a single DFD which is derived from the hierarchy of DFDs. To derive this single DFD we must be able to unabstract the decomposition of DFDs. Balancing will insure this is possible.

2.3.2 Ambiguities in the Data Dictionary

The DDE definitions of flow and store labels are based on a lexical description of value. The DDE syntax is equivalent to BNF notation for description of language syntax. This lexical description is based on undefined primitives and character strings. Furthermore, operations on flow and store data elements are limited to those described in P-Spec pseudo-code.
These limited facilities for representation of data objects are antiquated in comparison to the modern development of abstract data types and object-oriented programming. In Chapter 6 we develop a more modern, useful approach to the definition of flows and in Chapter 8 we consider the potential for similar definitions of stores.

2.3.3 Ambiguities in Process Specifications

P-Specs represent the bottom layer of the DFD hierarchy. The labels of in-flows and out-flows of a bubble specified by a P-Spec also appear in the P-Spec as variable parameter names. We question whether this is consistent with their use as flow labels. Are they naming values or structures (types)? Traditional methods for specifying the functional behavior of P-Specs are structured English, decision trees, and decision tables. Yourdon also includes a declarative style specification using preconditions and postconditions [66]. Decision trees and decision tables provide graphical notation for declarative style specification of actions. However, the traditional basis for all these tools is English which is inherently ambiguous. Furthermore, the use of structured English or any other pseudo-coded language usually results in the specification of an algorithm, which we have stated should be part of a design, not specification.

2.3.4 Ambiguities in Extensions to SA Specifications

Not only are the extensions to SA specifications ill-defined, but we question whether they are truly necessary. This is particularly true of the real-time extensions. In this thesis we do not address the development of a formal interpretation of ERDs, control bubbles, or control flows.
CHAPTER 3. ABSTRACT MODEL SPECIFICATIONS IN SPECS

3.1 Overview

In this chapter, we provide a general overview of SPECS [6, 7] and introduce 2 modifications to the SPECS language. SPECS is a formal abstract model specification language analogous to other abstract model languages like Z [28] and VDM [33]. SPECS was developed at Iowa State University for use in their Software Engineering course and as a research tool. Many of the examples in this chapter are drawn from course notes used at Iowa State University.

SPECS is used to define an abstract data type (ADT) which is composed of an abstract domain and a set of operations on elements of that domain. The ADT is then used as the basis for the design of a software system.

In this thesis we use the facilities of SPECS for describing abstract domains to model the syntax of SA specifications (and our enhancements to SA specifications). We also incorporate SPECS syntax and semantics directly into the syntax of SA specifications to provide for formal interpretation of SA components previously specified informally.

In this chapter we present a modified version of SPECS. The modifications include:

- extension of the primitive type string to allow for the direct specification of
string value restrictions using BNF notation;

- replacement of the language dependent syntax for specification of operations
  with a language independent syntax which facilitates ADT composition.¹

The general outline of a SPECS ADT specification is given in Figure 3.1. In
Section 3.2 we present the SPECS notation for the specification of an ADT domain.
In Section 3.3 we present the SPECS notation for the definition of constant values,
interface types, and an expression modularization feature. These components of a
SPECS ADT provide for extended formalisms we have found convenient and useful
for applying abstract model specifications to the development of software and for the
general improvement of readability of a specification. In Section 3.4 we present a
new notation for the specification of ADT operations that facilitates composition of
ADTs.

3.2 The ADT Domain

A SPECS ADT domain serves to define a set of valid instances of an abstract
data type. The ADT domain is usually specified in two parts—a source set and an
invariant property. The source set defines the set of all instances with the appropriate
abstract structure. The invariant property defines additional restrictions on the range
of valid values an instance of the domain may take on. Usually an ADT domain is
defined by a composition of abstract type declarations followed by a single invariant
property over instances of the ADT domain source set. However, to encourage the

¹We are currently considering extending the notation to include inheritance akin
to objected–oriented languages like Smalltalk and Eiffel.
Figure 3.1: General Form of a SPECS ADT

```
ADT:TypeName
  domain
    source set
    ...
    invariant
    ...
  definitions
    constants
    ...
  types
    ...
  expressions
    ...
  operations
    ...
```
use of English commentary for explanatory purposes, users can mix the presentation of abstract type declarations and invariant property restrictions in an ADT domain.

The type declarations which comprise the ADT are presented in a top-down order, i.e., the name of the ADT is defined first. An invariant property restriction may be presented whenever sufficient type declarations have been provided to allow type checking of the invariant property expression. Invariant properties may be stated over the entire ADT or individual types which comprise the ADT. All references to a particular abstract type name, implicitly include any restrictions stated in the domain section of the ADT.

3.2.1 The Domain Source Set

The source set component of the domain serves to define the abstract structure of elements of the domain. Abstract structure is defined using well-defined mathematical primitive and structured types. Associated with each type is a collection of mathematical operations defined on objects of that type.

The primitive types include mathematical integers, mathematical reals (with infinite precision), characters (any well-defined character set such as ASCII, denoted char), strings, booleans, user-defined enumerated types, and generic types. The structured types include set, sequence, tuple, and alternation. We also allow the definition of recursively defined types, however, we require that any element of the ADT domain be a finite object. The domain itself can define an infinite set. In the following sections we present the syntax for each type declaration, define the permissible operations on objects of that type, and define the constants of that type. We begin with the structured types.
3.2.1.1 Sets: A set defines an unordered collection of elements of a particular element type. Any instance of a set must be finite. A set cannot contain duplicate elements. A set type declaration has the following syntax:

\[ \text{SetTypeName} = \text{set of ElementTypeName}; \]

ElementTypeName can be any non-generic primitive type or a type name which is defined elsewhere in the source set. As we will see later, generic types must be named and thus cannot be used directly in the composition of structured types.

We use standard mathematical operations on sets and their elements. Let \( S_1 \) and \( S_2 \) be sets of some element type \( E \) and let \( e \) be of type \( E \). The following operations are provided on sets:

- union: \( S_1 \cup S_2 \) returns the set of all elements in either \( S_1 \) or \( S_2 \).
- intersection: \( S_1 \cap S_2 \) returns the set of all elements in both \( S_1 \) and \( S_2 \).
- subtraction: \( S_1 - S_2 \) returns the set of all elements in \( S_1 \) but not in \( S_2 \).
- equality: \( S_1 = S_2 \) returns true if \( S_1 \) and \( S_2 \) contain the same element values and otherwise returns false.
- inequality: \( S_1 \neq S_2 \) returns true if \( S_1 \) and \( S_2 \) do not contain the same element values and otherwise returns false.
- subset: \( S_1 \subseteq S_2 \) returns true if all element values in \( S_1 \) are also in \( S_2 \) and otherwise returns false.
- proper subset: \( S_1 \subset S_2 \) equivalent to \( (S_1 \subseteq S_2 \text{ and } S_1 \neq S_2) \).
superset: \( S_1 \supseteq S_2 \) equivalent to \( S_2 \subseteq S_1 \).

proper superset: \( S_1 \supset S_2 \) equivalent to \( S_2 \subset S_1 \).

membership: \( e \in S_1 \) returns true if \( e \) is an element in \( S_1 \) and otherwise returns false.

non-membership: \( e \notin S_2 \) returns true if \( e \) is not an element in \( S_2 \) and otherwise returns false.

cardinality: \( |S_1| \) returns the number of elements in \( S_1 \).

Given a predicate \( P(s) \) and a function \( F(s) \) whose result is a set we allow predicated forms of union (denoted \( \bigcup_{P(s)} F(s) \)) and intersection (denoted \( \bigcap_{P(s)} F(s) \)). Predicated union is interpreted as the union over all \( s \), for which \( P(s) \) is true, of \( F(s) \). Predicated intersection is interpreted similarly. We also extend any multiple grouping of relational operators to an \( n \)-ary form, e.g. \( S_1 = S_2 \subseteq S_3 \) is equivalent to \( S_1 = S_2 \wedge S_2 \subseteq S_3 \).

There are two ways to define set constants. One is to list a finite number of elements within set brackets. For example:

\[
\{ e_1, e_2, \ldots, e_n \}
\]

An element \( e_i \) can be any expression whose value is an element of the domain of \text{ElementTypeName}. The empty set is denoted \( \{ \} \). A second method to define set constants is to predicate the membership of elements. For example, given \( P(e) \) is a first order predicate in which all variables in \( e \) are free variables in \( P \):

\[
\{ e \mid P(e) \}
\]
is read the set of all e such that \( P(e) \) is true.

3.2.1.2 Sequences: A sequence defines an ordered collection of items of a particular item type. Any instance of a sequence must have a finite number of items. Duplicate items are allowed since they can be distinguished by position in the sequence. A sequence type declaration has the following syntax:

\[
\text{SequenceTypeName} = \text{sequence of ItemTypeName};
\]

ItemTypeName can be any non-generic primitive type or a type name which is defined elsewhere in the source set.

Let \( S_1 \) be a sequence of items \( x_1, x_2, \ldots, x_n \) of some type I in the given order, where \( n \) is a non-negative integer. Let \( S_2 \) be a sequence of items \( y_1, y_2, \ldots, y_m \) of type I in the given order, where \( m \) is a non-negative integer. Let \( i \) be of type I. We denote a list of items as a sequence by enclosing the list in angled brackets; e.g., \( S_1 \) is denoted \(<x_1, x_2, \ldots, x_n>\). Operations on sequences are as follows:

- **concatenation:** \( S_1 \| S_2 \) returns the sequence \(<x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m>\).

- **header:** \( \text{head}(S_1) \) returns the sequence \(<x_1, x_2, \ldots, x_{n-1}>\), if \( n \geq 1 \) and otherwise results in an error.

- **trailer:** \( \text{tail}(S_1) \) returns the sequence \(<x_2, x_3, \ldots, x_n>\), if \( n \geq 1 \) and otherwise results in an error.

- **subsequence:** \( \text{subseq}(S_1, a, b) \) returns the sequence \(<x_a, x_{a+1}, \ldots, x_{a+b-1}>\) if \( 1 \leq a \leq a + b \leq n \) and otherwise results in an error.

- **first:** \( \text{first}(S_1) \) returns \( x_1 \) if \( n \geq 1 \) and otherwise results in an error.
last: \( \text{last}(S1) \) returns \( x_n \) if \( n \geq 1 \) and otherwise results in an error.

index: \( \text{index}(S1, a) \) returns \( x_a \) if \( 1 \leq a \leq n \) and otherwise results in an error. (We often abbreviate \( \text{index}(S1, a) \) as \( S1_a \) whenever the abbreviated form results in no ambiguity.

equality: \( S1 = S2 \) returns true if \( n = m \) and \( x_k = y_k \) for all \( 1 \leq k \leq n \) and otherwise returns false.

inequality: \( S1 \neq S2 \) returns true if \( n \neq m \) or there exists \( k, 1 \leq k \leq n \) such that \( x_k \neq y_k \) and otherwise returns false.

membership: \( i \in S1 \) returns true if there exists \( k, 1 \leq k \leq n \) and \( i = \text{index}(S1, k) \) and otherwise returns false.

non-membership: \( i \notin S1 \) returns true if for all \( k, 1 \leq k \leq n \) implies \( i \neq \text{index}(S1, k) \) and otherwise returns false.

length: \( \text{length}(S1) \) returns \( n \).

Sequence constants are defined by listing a finite number of item expressions inside angled brackets. The empty sequence is denoted \(< >\).

3.2.1.3 Tuples: A tuple defines a finite mapping of names to items not necessarily of the same type. Although we generally refer to a tuple as an unordered collection of items, we use the order given in the type definition to define the semantics of operations and constants. A tuple containing \( n \) items, \( n \geq 0 \), is referred to as an \( n \)-tuple. An \( n \)-tuple type declaration has the following syntax:
TupleTypeName = n-tuple(ItemName1:TypeName1, ItemName2:TypeName2, ... ItemNameN:TypeNameN);

Each ItemName_i must be unique. An TypeName_i may either be a set or sequence type declaration, a non-generic primitive type, or a type name which is defined elsewhere in the source set.

The only common operation defined on n-tuples is the referencing of a named item. We use a functional notation to reference items. The i_th item of an n-tuple T is:

ItemName_i(T)

Two tuples of the same TupleTypeName may also be compared for equality (=) and inequality (≠) with the usual semantics. Tuple constants are constructed by listing items enclosed with parenthesis optionally followed by a colon and the appropriate TupleTypeName. For example, consider the tuple type Rational defined as follows:

Rational = 2-tuple(Numerator:integer, Denominator:integer);

A Rational constant with numerator 5 and denominator 8 is defined:

(5,8) : Rational

The TupleTypeName is required to distinguish between tuple constants of the same size, but different TupleTypeNames. We drop the TupleTypeName when it can be
easily inferred. Any TupleTypeName declared as a 0-tuple has a domain of size one whose only element is the special value Null.

3.2.1.4 Alternation: An alternation represents a union of the domains of two or more types. No two of the alternative types can be type name equivalent. An object of an alternative type has one of a known set of alternative types. An alternative type is declared as follows:

\[
\text{AlternativeTypeName} = \text{AlternativeType}_1 \mid \ldots \mid \text{AlternativeType}_n
\]

Where \( n \geq 2 \) and for all \( i, \) for all \( j, 1 \leq i < j \leq n \) implies \( \text{AlternativeType}_i \neq \text{AlternativeType}_j. \)

The only operations on alternative typed objects is distinction of their alternative type. Determination of alternative type is based solely on type name equivalence and not on structural equivalence. Constants which are structurally equivalent to more than one alternative must be injected into one of the alternative types. Injection is accomplished by suffixing the constant with a colon and type name, e.g., 3:integer.

We define two relational operators is-of-type (denoted ?) and is-not-of-type (denoted /). Given an object \( A \) declared of type AlternativeTypeName,

\[
A\text{?}\text{AlternativeType}_i,
\]

\( 1 \leq i \leq n, \) is true if object \( A \) is an element of the domain defined by \( \text{AlternativeType}_i \) and is otherwise false. \( A\text{/}\text{AlternativeType}_i \) is true if \( A \) is not an element of the domain defined by \( \text{AlternativeType}_i \) and is otherwise false. Note that \( A \) must be an element of exactly one of the alternative type domains.

In any specification expression, the alternative type of an alternatively typed object \( A \) must be determined before operations allowed on that alternative type can
be applied to $A$. For example, assume we wanted to assert that the size of an object $S$ is equal to $n$ and $S$ was declared as having type SetOrSequence defined as follows:

SetOrSequence = SetOfReals | SequenceOfIntegers;
SetOfReals = set of real;
SequenceOfIntegers = sequence of integer;

The following expression shows how we must first determine whether $S$ is a set or a sequence before applying the size operation of sets or sequences.

$$(A : ?\text{SetOfReals} \Rightarrow |A| = n) \land
(A : ?\text{SequenceOfInteger} \Rightarrow length(A) = n)$$

3.2.1.5 Recursive Types: Alternation provides the ability to declare recursively defined types. This is best illustrated by an example. Consider the following declaration of the type BinaryTree:

BinaryTree = NonEmptyBinaryTree | EmptyBinaryTree;
NonEmptyBinaryTree = 3-tuple(Root:NodeType,
LeftChild:BinaryTree,
RightChild:BinaryTree);
EmptyBinaryTree = 0-tuple( );
NodeType = ...

A 0-tuple represents a tuple structure which contains no information. The information denoted by an object of type 0-tuple is only its type, i.e., knowing a BinaryTree $B$ is-of-type EmptyBinaryTree provides no more information than its type. Although the BinaryTree declaration contains the potential of representing
an infinitely recursive structure, we require that any element of type BinaryTree be finite. Thus, eventually both the LeftChild and RightChild of a NonEmptyBinaryTree must be EmptyBinaryTrees. The set of all objects in the domain of BinaryTree is infinite, but every object itself is finite.

Recursion does present the possibility of confusion when distinguishing alternative types. For example, modeling context-free grammars with type declarations can create unstructured recursion. This is illustrated by the following type declaration of an expression syntax. Note that there is no lexical representation of the parenthesis in a ParenthesizedExpr.

\[
\text{Expression} = \text{SimpleExpr} \mid \text{BinaryExpr} \mid \text{ParenthesizedExpr};
\]

\[
\text{SimpleExpr} = \text{string};
\]

\[
\text{BinaryExpr} = 3\text{-tuple} (\text{Operator}: \text{OperatorType}, \\
\text{Operand1}: \text{Expression}, \\
\text{Operand2}: \text{Expression});
\]

\[
\text{ParenthesizedExpr} = \text{Expression};
\]

\[
\text{OperatorType} = \ldots
\]

Consider the recursive expression definition, given in Figure 3.2, which defines the value an object of type Expression. There is no representation of "getting inside of" the parenthesis in the evaluation of a ParenthesizedExpr. In the third conjunction, is Evaluate being applied to the original Expression type object or is it being applied to a ParenthesizedExpr type object? There are three potential solutions this problem:
define Evaluate\( (E: \text{Expression}) \) as ResultType such that
\begin{align*}
(E:\text{SimpleExpr} \Rightarrow & \quad \text{Evaluate} = \text{ValueBoundTo}(E)) \\
\land (E:\text{BinaryExpr} \Rightarrow & \quad \text{Evaluate} = \text{Apply}(\text{Operator}(E), \text{Evaluate}(\text{Operand1}(E)), \text{Evaluate}(\text{Operand2}(E)))) \\
\land (E:\text{ParenthesizedExpr} \Rightarrow & \quad \text{Evaluate} = \text{Evaluate}(E))
\end{align*}

Figure 3.2: A Recursive Expression Definition

1. Adopt fixed point semantics [54]. If we assume that Evaluate is non-deterministically applied to either the original Expression type object or to a ParenthesizedExpr type object, then fixed point semantics can be used to determine the meaning of the expression definition.

2. Adopt an implicit scoping mechanism. Suppose in an implication the antecedent contains \( X\text{?XType} \). Then in the consequent of the implication all references to \( X \) are interpreted as references to an object of type \( X\text{Type} \). This can lead to further confusion if the antecedent contains more than one is-of-type operator applied to the same object.

3. Require all recursion in type declarations to be enclosed within another structured type. In the above example, we need to enclose ParenthesizedExpr's use of Expression type. This can be done by structuring the Expression type use in a 1-tuple as follows:

\[
\text{ParenthesizedExpr} = \text{1-tuple}(\text{Operand}: \text{Expression});
\]
The third conjunction of $Evaluate$ is then replaced by

$$\land (\text{ParenthesizedExpr} \Rightarrow$$

$$Evaluate = Evaluate(\text{Operand}(E)))$$

which makes the "getting inside of" the parenthesis explicit.

For simplicity we adopt the third solution and require recursive type declarations to be enclosed within another structured type.

### 3.2.1.6 Primitive Types:
The SPECS primitive types provide the foundation for the composition of structured types. The primitive types are described as follows:

- **integer**: The domain of integers is unbounded. We assume the use of any standard arithmetic operations and relations. SPECS syntax will also allow the definition of any non-standard operations.

- **real**: The domain of reals is unbounded and provides infinite precision operations on reals. Any standard arithmetic operations and relations may be used.

- **char**: The domain of char can be defined as any finite character set. The default is the ASCII character set. If another character set is required it must be explicitly defined as follows:

  $$\text{char} = \{ \Gamma, \Delta, \Theta, \Lambda, \Xi, \Pi, \Sigma, \Upsilon, \Phi, \Psi, \Omega \}$$

Operations include ord, chr, pred, and succ as defined by Pascal [32] and the standard relational operations ($<$, $\leq$, $>$, $\geq$, $=$, $\neq$). Constants can be denoted by enclosing a single character in either double quotes or single quotes.
string: The domain string is modeled as a sequence of char. Operations include all operations defined on sequences, where subseq is renamed substring, with the addition of relational operations (\(<, \leq, >, \geq\)) which define a standard lexicographic ordering.

lexical strings: To support the modeling of syntactic objects we extend SPECS with Backus Nauer Form (BNF) description of subsets of the domain of strings. A type declaration of the following form defines a domain of all strings that are derivable from a BNF definition of Start. The BNF description of Start provides an invariant-like restriction on objects of type string.

\[
\text{LexicalStringType} = \text{BNF}[\text{Start}];
\]

A BNF description of Start must then be provided. Table 3.1 provides a list of the meta-symbols used to define a BNF description. Terminal symbols can be indicated either by boldfont or by enclosing them in single quotation marks.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ab)</td>
<td>a concatenated with b</td>
</tr>
<tr>
<td>(a</td>
<td>b)</td>
</tr>
<tr>
<td>({a})</td>
<td>zero or more occurrences of a</td>
</tr>
<tr>
<td>({a}^+)</td>
<td>one or more occurrences of a</td>
</tr>
<tr>
<td>([a])</td>
<td>zero or one occurrence of a</td>
</tr>
<tr>
<td>(a(b</td>
<td>c))</td>
</tr>
</tbody>
</table>

Table 3.1: BNF Meta-symbols

boolean: The domain of boolean is the set \{true, false\}. Operations include logical-and (denoted \(\land\)), logical-or (denoted \(\lor\)), logical-implies (denoted \(\Rightarrow\)), logical-equivalence (denoted \(\equiv\)), and logical-negation (denoted \(\neg\)).
user-defined enumerated types: User defined enumerated types allow for the creation of any finite set of named values. We use the syntax, semantics, and operations defined by Pascal [32].

generic, equality-generic, and ordered-generic: A generic type can be used to generalize abstract structured types. For example, an ADT Stack can be defined as a sequence of generic objects. The particular type of objects stacked are then defined by the user of the ADT Stack. We do require that generic types be named. For example ItemType names a generic in the following:

\[
\text{Stack} = \text{sequence of ItemType;}
\]
\[
\text{ItemType} = \text{generic;}
\]

We provide three forms of the generic type: one simply called generic with no implied operations; one called equality-generic with operations equality (\(=\)) and inequality (\(\neq\)); and one called ordered-generic with the additional operations less than (\(<\)), less than or equal (\(\leq\)), greater than (\(>\)), and greater than or equal (\(\geq\)). Comparison of equality-generic and ordered-generic types is only allowed for type name equivalent generics.

Instantiation of generic ADTs is accomplished by a type name assignment when the generic ADT is used in a declaration. Instantiation is not required when using a generic ADT. Instantiation may occur in an abstract declaration or an implementation declaration in either a type, variable, or parameter declaration. For example, declaring a variable \(\text{IntStk}\) as a stack of integers would have the following syntax:

\[
\text{IntStk} = \text{Stack of Integers;}
\]
IntStk : Stack[ItemType← integer];

All relational operators are extended to n-ary operators as discussed previously. Any type may also be named by providing a declaration which defines the type name, i.e., "IntType = integer;" defines the type name IntType as having the same domain as integers.

3.2.2 The Domain Invariant

The second component of an ADT domain provides the ability to further restrict the domain provided by the source set, based on domain element value. Assume XType is a type name defined in an ADT source set and assume \( P(X) \) is a first order predicate in which \( X \), a variable of type XType, is a free variable. An invariant property which restricts the domain of all occurrences of objects of type XType to include only those for which \( P \) is true would have the following syntax:

\[
\text{(for any } X \text{ of type XType):} \\
P(X)
\]

Within the invariant property the name XType refers to the structural definition of XType provided by a type declaration in the source set. When XType is used elsewhere in the ADT specification, XType refers to the set of all objects in the source set of XType that satisfy the predicate \( P \).

Recall the tuple type declaration example for Rational. Elements of the source set Rational whose Denominator equals zero do not represent valid rational numbers. To express this restriction we would use the following invariant property:
(for any $R$ of type Rational):

$$\text{Denominator}(R) \neq 0$$

The operators we use to form first order predicates include or ($\lor$), and ($\land$), implies ($\Rightarrow$), equivalence ($\equiv$) and negation ($\neg$). Precedence in decreasing order is $\neg$, $\land$, $\lor$, $\Rightarrow$, $\equiv$, and parenthesis can be used for grouping to override precedence. We use the following notation for quantified expressions:

$$\text{list-of-quantified-variables}[P]$$

Each variable is quantified by either for-all ($\forall$), there-exists ($\exists$), or there-exists-a-unique ($\exists!$).\(^2\) Quantified variables may be optionally followed by a colon and a type name or a non-generic primitive type to restrict their domain and improve readability. The scope of a quantified variable is bounded by the next set of square brackets. Given predicates $P(x)$ and $Q(x)$ we also allow predicated forms of and (denoted $\left( \land Q(x) \right)$) and or (denoted $\left( \lor Q(x) \right)$). Predicated and is interpreted as the conjunction, over all $x$ for which $P(x)$ is true, of $Q(x)$. Predicated or is defined similarly.

### 3.3 ADT Definitions

The definitions section of an ADT provides for the definition of abstract constants, type definitions, and modularization of first order predicate expressions.\(^2\) $\exists!$ is used to assert the unique existence of an element from the universe of discourse, i.e., $\exists! x [P(x)]$ asserts that there is exactly one and only one $x$ in the universe of discourse for which $P(x)$ is true.
3.3.1 Constant Definitions

Constant definitions may be generalized to any constant that satisfies a particular first order predicate using the keyword any. For example, the following constant declarations defines StackSize as any positive integer and RatZero as any Rational whose numerator is zero:

\[\text{StackSize} = \text{any:integer such that StackSize} > 0;\]
\[\text{RatZero} = \text{any:Rational such that Numerator(RatZero)} = 0;\]

Constants values may also be explicitly stated. For example:

\[\text{QueueLength} = 10:\text{integer};\]
\[\text{OneThird} = (1,3):\text{Rational};\]

3.3.2 Type Definitions

Type definitions provide domains for operation input parameter types, operation result types, or quantified variables in expressions. These type definitions have the same format as the ADT domain type declarations and invariants. They are separated from the ADT domain because they do not contribute to the definition of the ADT domain. For example:

constants

\[\text{ExitChoice} = \text{any:integer such that ExitChoice} > 0;\]

types
MenuChoice = integer;
(for any $M$ of type MenuChoice):
$1 \leq M \leq \text{ExitChoice}$

### 3.3.3 Expression Definitions

Expression definitions provide for a parameterized and named modularization of first order predicate expressions. The general syntax of an expression definition is as follows:

\[
\text{define } \textit{ExpressionName}(\textit{Parameter}_1: \textit{ParameterType}_1, \\
\vdots \\
\textit{Parameter}_n: \textit{ParameterType}_n) \text{ as ResultType} \\
\text{such that} \\
\textit{P}(\textit{ExpressionName}, \textit{Parameter}_1, \ldots, \textit{Parameter}_n)
\]

$P$ represents a first order predicate in which $\textit{ExpressionName}$, $\textit{Parameter}_1$, through $\textit{Parameter}_n$ appear as free variables. ResultType may be either a set or sequence declaration, a primitive type, or the name of a defined type.

User defined, infix, binary operators can be defined using an expression definition, by placing the $\textit{ExpressionName}$ in an infix notation between two parameters in an expression definition. For example, in Chapter 5 we redefine infix binary equality ($=$) for a type ModifiedLabel using the following expression definition header:

\[
\text{define } (X:\text{ModifiedLabel})"="(Y:\text{ModifiedLabel}) \text{ as boolean} \\
\vdots
\]
Naming of expression modularizations allows for recursive expression definitions. For example, the following expression definition defines the height of a BinaryTree. We assume the \texttt{max} defines the maximum of two integer or real values.

\[
\text{define } \textit{Height}(B: \text{BinaryTree}) \text{ as integer such that } \\
(B: ?\text{EmptyBinaryTree} \Rightarrow B: \text{BinaryTreeHeight} = 0) \\
\land (B: ?\text{NonEmptyBinaryTree} \Rightarrow \\
\text{Height} = 1 + \text{max}(\text{Height}(\text{LeftChild}(B)), \text{Height}(\text{RightChild}(B))))
\]

Expression definitions do not represent computational operations, i.e., they are not implemented. For this reason they are not restricted to finite sized objects. For example, an expression definition which defined the set of all paths in a flowgraph representation of an imperative program, may define an infinite set of paths, if the program contains a generalized looping structure.

### 3.4 ADT Operations

The collection of ADT operations provide the only means by which an instance of the ADT domain can be created, modified, or queried by a program. Operations are defined using a precondition and postcondition. Operations have the following syntax:

\[
\text{operation } \textit{OperationName}(\text{Parameter}_1, \ldots, \text{Parameter}_{1m_1}: \text{ParameterType}_1; \\
\text{Parameter}_1, \ldots, \text{Parameter}_{2m_2}: \text{ParameterType}_2; \\
\vdots \\
\text{Parameter}_N, \ldots, \text{Parameter}_N: \text{ParameterType}_N)
\]
:ResultType;

pre: \( P(\text{Parameter} \, 1, \ldots, \text{Parameter} \, N) \)

post: \( Q(\text{Operation} \, \text{Name}, \text{Parameter} \, 1, \ldots, \text{Parameter} \, N) \)

All parameters are "passed" by value. ResultType may be any type definition or the name of a defined type. \( P \) and \( Q \) represent first order predicate calculus expressions in which the given variables appear free. In the case where an operation has no parameters, the parenthesis enclosing the parameterlist are omitted.

The precondition \( P \) must be true prior to execution of the operation. The postcondition \( Q \) is guaranteed to be true upon completion of the operation. Three types of operations can be identified based on appearance of the ADT domain type in the parameter list or as the ResultType. Table 3.2 identifies the three types.

<table>
<thead>
<tr>
<th>ADT domain type appears in parameter list</th>
<th>ADT domain type appears as ResultType</th>
<th>Type of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>yes</td>
<td>create</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>modify</td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
<td>query</td>
</tr>
</tbody>
</table>

Certain properties must hold for the set of operations defined by an ADT. They are:

completeness: the ability to generate any element in the ADT domain.

testability: the non-trival preconditions must be testable by the set of ADT operations.
maintenance of invariant: any item generated must satisfy the domain invariant.

usability: the set of operations must provide a useful solution to the requirements of the software system. This property is difficult to prove.
CHAPTER 4. AN ABSTRACT MODEL FORMALIZATION OF STRUCTURED ANALYSIS SPECIFICATIONS

4.1 Overview

In this chapter we present an abstract model formalization of a traditional SA specification. If we intended to provide specifications for operations to construct and modify SA specifications, then our abstract model could be construed as a specification of a CASE tool for constructing SA specifications. This is not our intent.

Our intent is to provide a rigorous framework for interpreting the meaning of a SA specification. For this interpretation to be unambiguous, and thus serve as an effective communication tool, our framework for SA specifications must be precise. Our abstract model will define a domain containing only syntactically correct SA specifications. The abstract model of an SA specification will allow us to describe the precise semantics associated with the SA modeling tools, i.e., DFD, P-Specs, DDE, etc.

To achieve precise interpretations, we need to replace the underlying informal structures of SA specifications with formal syntactic structures. Recall for example, primitive process specifications (P-Specs) which are traditionally given in English or at best a structured pseudo-code. English specifications are inherently ambiguous. Structured pseudo-code can also be ambiguous or can provide an algorithmic strategy
which says how a function is to be achieved rather than just what the function is.¹ We rely on the abstract model specification techniques to achieve this precision. By incorporating abstract model specification techniques in SA specifications we provide both mathematical preciseness and implementation abstraction.

In our evaluation of traditional SA specification techniques, we have occasion to consider the expressive value of a technique. By expressive value, we mean the ease with which we, as humans, can express and manage an SA specification. This is not the same as, but also not orthogonal to, computational expressiveness. Computational expressiveness relates to the number and kinds of different programs that can be specified by a particular technique. In the rest of this thesis, our use of the term expressive relates to the notion of human expressiveness and manageability, not computational expressiveness.

In Section 4.2 we provide a SPECS abstract model domain formalization of a traditional SA specification. Our abstract model formalization is based on the graphical and textual syntax of an SA specification. Thus our mathematical model will describe the valid graphical and textual structures of an SA specification, but not what the structures mean. In particular, we focus our attention on the syntax of the DFD hierarchy, the DDE definitions, and the correlation between DFDs and DDEs. In effect, our abstract model will describe a syntactically correct SA specification in much the same way a compiler accepts a program in a particular language. If all the identifiers have been declared correctly, the begins and ends match, etc., then the program is syntactically correct. However, the meaning of the program is based on

¹The algorithmic description of how a function's result is achieved is a design decision and should not normally appear as part of the specification.
its execution (or generated machine code).

4.2 The SA Model

We begin by presenting a collection of abstract type definitions that define the syntactic domain of SA specifications. We also include abstract invariant properties to restrict the domain to SA specifications that are syntactically well-defined, i.e., represent graphic displays of completed SA specifications.

Consider the example SA specification in Figure 4.1. An SA specification separates a system into two components; the functional component is described by the hierarchy of DFDs and P-Specs; the data component is described by the DD of flow and store labels. One alternative for defining the domain of SA specifications would be to model the hierarchy of DFDs directly by a tree-like domain. An SA specification would be defined as a context diagram and a DD. Each bubble in a DFD would be defined by either another DFD or a P-Spec. Another alternative is to consider the collection of DFDs and P-Specs contained in the hierarchy as a set of processes independent of the hierarchical ordering. In this case an SA specification would be described as a set of processes and a DD. The hierarchy of DFDs would not be directly modeled by the abstract domain, rather the existence of the hierarchy would be asserted by an invariant property. We choose the second alternative principally to facilitate the statement of invariant properties which apply to all processes. Using this structure, these domain invariants are easier to state and are thus more readable.

We define an SA specification in Type Definition 1. The model label provides a name for the system we are specifying and signifies the label of the context DFD. The terminators define the set of entities outside the scope of the system. The processes
Figure 4.1: Example SA Specification
define the collection of DFDs and P-Specs that make up the DFD hierarchy. The DD provides specifications for the data objects represented by the flows and stores appearing in DFDs. We do not include the specification of P-Specs in the DD as suggested by De Marco [18]. This is because processes are not data, but rather data manipulators and again, for the simple reason that it is easier to state the invariant properties in this manner.

**Type Definition 1**

\[
\text{SA-Spec} = 4\text{-tuple}( \text{ModelLabel} : \text{Label}, \text{Terminators} : \text{set of TermType}, \text{Processes} : \text{set of ProcessType}, \text{DataDictionary} : \text{set of DDE});
\]

Example 1 on page 62 provides a partially completed abstract object that represents the example SA specification shown Figure 4.1. As we introduce additional abstract type structures used to define each SA specification component, we will provide the missing pieces to complete the example abstract object. The example is abstract in the sense that the labels, e.g., Bubble0, Flow1, etc., are serving only as placeholders. For example, eventually we will require that the label represented by the placeholder Bubble0 equal the label represented by the placeholder DFD0.

**Example 1**

\[
(\text{Name},
\{\text{Terminator1, Terminator2, Terminator3}\},
\{(\text{Name},\ldots,)(\text{DFD0},\ldots),(\text{DFD3},\ldots),(\text{P-Spec1},\ldots),
(\text{P-Spec2},\ldots),(\text{P-Spec3.1},\ldots),(\text{P-Spec3.2},\ldots),(\text{P-Spec3.3},\ldots)\},
(\text{Flow1},\ldots),(\text{Flow1.1},\ldots),(\text{Flow1.2},\ldots),(\text{Flow2},\ldots),(\text{Flow3},\ldots),
(\text{Flow4},\ldots),(\text{Flow5},\ldots),(\text{Flow6},\ldots),(\text{Flow7},\ldots),(\text{Flow7.1},\ldots),
(\text{Flow7.2},\ldots),(\text{Flow8},\ldots),(\text{Flow9},\ldots),(\text{Flow10},\ldots),(\text{Store1},\ldots)\})
\]
Traditionally, terminators are specified only by a descriptive name, reflected by Type Definition 2. When we examine terminators more closely in Chapter 8 we will modify the definition of TermType to reflect a more rigorous characterization of terminator behavior.

**Type Definition 2**

```
TermType = Label;
```

### 4.2.1 The DFD Hierarchy

The processes in an SA specification are divided into two groups; those specified by a DFD and primitive processes specified by a P-Spec. This is reflected in Type Definition 3.

**Type Definition 3**

```
ProcessType = DFD | P-Spec;
```

We include within the syntactic structure of a DFD those features that are visible in the graphic representation of a DFD. For example, flows will have labels, but the structure of the data represented by a flow will be given in the DD. Type Definition 4 defines the structure of a DFD. Each DFD is uniquely identified by a process label. We adopt Yourdon/DeMarco diagraming techniques for DFDs as shown in Figure 4.2 [18, 66]. Bubbles are drawn as labeled circles. Stores are drawn as two parallel lines with a modified label between. (A *modified* label differs syntactically and semantically from a non-modified label. The syntactic difference will be provided
later in this chapter. The particular semantics of modified labels will be provided in Chapter 5.) Both modified and non-modified labels serve to uniquely identify DFD components and, when the context is clear, we refer to both types of labels as just labels. Terminators, which appear only in the context diagram (the DFD at the top of the hierarchy) are drawn as labeled rectangles. Flows are drawn as directed edges labeled by modified labels. The entire DFD is enclosed in a dotted line rectangle with rounded corners. The process label for a DFD usually appears in the upper left corner. Adding the structure of DFDs to the example SA specification results in the abstract object provided in Example 2 on page 65.

**Type Definition 4**

\[
\text{DFD} = 4\text{-tuple}(\ \text{ProcessLabel} : \text{Label}, \\
\text{Bubbles} : \text{set of Label}, \\
\text{Stores} : \text{set of ModifiedLabel}, \\
\text{Flows} : \text{set of FlowType});
\]

![Diagram of DFD components](image)

**Figure 4.2: DFD Diagraming Techniques**

Type Definition 5 defines the non-modified labels attached to processes, bubbles, and terminators. Invariant Property 1 assures that process, bubble, and terminator
Example 2

(ModelName,
{Terminator1,Terminator2,Terminator3},
{(ModelName,
 {Bubble0},
 { },
{(Flow1,...),(Flow2,...),(Flow3,...)}),
(DFD0,
 {Bubble1,Bubble2,Bubble3},
 {Store1},
{(Flow1.1,...),(Flow1.2,...),(Flow2,...),(Flow3,...),
 (Flow4,...),(Flow5,...),(Flow6,...),(Flow7,...)})},
(DFD3,
 {Bubble3.1,Bubble3.2,Bubble3.3},
 { },
{(Flow2,...),(Flow6,...),(Flow7.1,...),(Flow7.2,...),
 (Flow8,...),(Flow9,...),(Flow10,...)}),
(P-Spec1,...),(P-Spec2,...),(P-Spec3.1,...),(P-Spec3.2,...),(P-Spec3.3,...)},
{(Flow1,...),(Flow1.1,...),(Flow1.2,...),(Flow2,...),(Flow3,...),
 (Flow4,...),(Flow5,...),(Flow6,...),(Flow7,...),(Flow7.1,...),
 (Flow7.2,...),(Flow8,...),(Flow9,...),(Flow10,...),(Store1,...)})}
labels are non-empty.

**Type Definition 5**

Label = string;

**Invariant Property 1**

(for any $L$ of type Label): $L \neq "un"

A flow in a DFD is defined by its source, destination, and label. Type Definition 6 defines the syntactic structure of a flow. A flow can have a bubble, terminator, or store as an actual destination or source. Some flows, called off-page flows, have sources and destinations that exist at a higher level in the DFD hierarchy. For example, Flow1.1 and Flow2.2 in Figure 4.1 are off-page flows. Flows can also be drawn as converging or diverging as shown in Figure 4.3.

Converging and diverging flows may have multiple labels and we distinguish each convergent or divergent part of a flow as a separate flow. For example, in Figure 4.3 (a) we would distinguish three separate flows labeled X, Y, and Z. In Figure 4.3(b) we would distinguish three separate flows all labeled Z.

**Type Definition 6**

FlowType = 3-tuple( FlowLabel : ModifiedLabel, Source : EndPoint, Destination : EndPoint);

---

2The labels X, Y, and Z are abstractions of modified labels, i.e., they represent place holders for modified labels.
To represent this multitude of possible sources and destinations we characterize a flow end point in Type Definition 7. Bubbles and terminators serving as flow end points are identified by their associated label. Similarly, stores serving as end points are identified by their associated modified label. A source or destination defined as off-page contains no information other than its type as defined by Type Definition 8. A source or destination that represents a point of convergence or divergence of several flows is identified by an integer in Type Definition 9. We generalize the reference to a convergence or divergence point as simply a vergence point. The integer value associated with a particular vergence point provides a unique reference for that vergence point. Vergence points are unique only with respect to a particular DFD.

Adding the structure of FlowTypes to the example SA specification results in the abstract object provided in Example 3 on page 69.

\[\text{Figure 4.3: Converging and Diverging Flows}\]

\[\begin{array}{cc}
\text{Converging Flows} & \text{Diverging Flows} \\
\begin{array}{c}
X \\
Y \\
Z \\
(a)
\end{array} & \begin{array}{c}
X \\
Z \\
Y \\
(c)
\end{array} \\
\begin{array}{c}
Y \\
Z \\
(b)
\end{array} & \begin{array}{c}
Z \\
(d)
\end{array}
\end{array}\]

\[\text{^The syntactic difference between modified labels and unmodified labels is presented subsequently.}\]
Type Definition 7

\[ \text{EndPoint} = \text{Label} \mid \text{ModifiedLabel} \mid \text{OffPage} \mid \text{Vergence}; \]

Type Definition 8

\[ \text{OffPage} = \text{0-tuple}(\ ); \]

A point of convergence must have at least two flows into it, called converging flows, and exactly one flow out of it, called the convergent flow. A point of divergence must have exactly one flow into it, called the diverging flow, and at least two flows out, called divergent flows.\(^4\) This restriction is characterized by Invariant Property 2.

Several invariant properties restrict the manner in which DFD objects can be connected by flows. For example, two stores cannot be directly connected by a flow. Convergent and diverging flows complicate this issue, since a convergent flow’s source can be construed as the set of all associated converging flow sources. Similarly, a diverging flow’s destination can be construed as the set of all associated divergent flow destinations. Thus the actual source and actual destination of a particular flow can be described as a set of bubbles, stores, and terminators. For example, if three flows, whose individual sources are a bubble, store, and terminator, converge then the convergent flow’s source is actually the set containing the bubble, store, and terminator labels. We do not extend this characterization of actual flow source and destination across DFD bubble decomposition. To express the actual source and destination across DFD bubble decomposition. To express the actual source and

\(^4\) The easy way to remember “vergence” flow names is “verging” going in and “vergent” coming out of a vergence point.
Type Definition 9

Vergence = integer;

Example 3

(ModelName,
 {Terminator1, Terminator2, Terminator3},
 {ModelName,
  {Bubble0},
  {}},
 {Flow1, Terminator1, Bubble0), (Flow2, Terminator2, Bubble0),
  (Flow3, Bubble0, Terminator3))},
(DFD0,
 {Bubble1, Bubble2, Bubble3},
 {Store1},
 {Flow1.1, Null, Bubble1), (Flow1.2, Null, Bubble1), (Flow2, Null, Bubble3),
  (Flow3, Bubble2, Null), (Flow4, Bubble1, Store1), (Flow5, Store1, Bubble2),
  (Flow6, Bubble1, Bubble3), (Flow7, Bubble1, 0), (Flow7, Bubble3, 0),
  (Flow7, 0, Bubble2}),
(DFD3,
 {Bubble3.1, Bubble3.2, Bubble3.3},
 {},
 {Flow2, Null, Bubble3.2), (Flow6, Null, Bubble3.1), (Flow7.1, Bubble3.3, Null),
  (Flow7.2, Bubble3.2, Null), (Flow8, Bubble3.1, Bubble3.3), (Flow9, Bubble3.1, Bubble3.2),
  (Flow10, Bubble3.2, Bubble3.3)),
(P-Spec1,..., (P-Spec2,..., (P-Spec3.1,..., (P-Spec3.2,..., (P-Spec3.3,...)),
{Flow1,..., (Flow1.1,..., (Flow1.2,..., (Flow2,..., (Flow3,...,)
  (Flow4,..., (Flow5,..., (Flow6,..., (Flow7,..., (Flow7.1,...,
  (Flow7.2,..., (Flow8,..., (Flow9,..., (Flow10,..., (Store1,...))})}
Invariant Property 2

(for any $P$ of type DFD)

$\forall n : \text{integer}[\]

\[
\begin{align*}
&\left|\left\{ f \mid f \in \text{Flows}(P) \land \text{Destination}(f) \land \text{Vergence} \land \text{Destination}(f) = n \right\}\right| \geq 2 \\
&\land \left|\left\{ f \mid f \in \text{Flows}(P) \land \text{Source}(f) \land \text{Vergence} \land \text{Source}(f) = n \right\}\right| = 1 \\
&\lor \left|\left\{ f \mid f \in \text{Flows}(P) \land \text{Destination}(f) \land \text{Vergence} \land \text{Destination}(f) = n \right\}\right| = 1 \\
&\land \left|\left\{ f \mid f \in \text{Flows}(P) \land \text{Source}(f) \land \text{Vergence} \land \text{Source}(f) = n \right\}\right| \geq 2 \\
&\lor \neg \exists f[f \in \text{Flows}(P) \land (\text{Source}(f) \land \text{Vergence} \land \text{Source}(f) = n \\
&\lor \text{Destination}(f) \land \text{Vergence} \land \text{Destination}(f) = n)]
\end{align*}
\]

To express the restrictions on the actual sources and destinations of flows, we use Expression Definitions 3 and 4 which describe the set of actual sources and actual destinations of any particular flow in a DFD. We describe the actual source of an
off-page in-flow and the actual destination of an off-page out-flow by a set containing the DFD process label. The process of connecting the flows into the parent bubble with the off-page flows in the child DFD is referred to as flow balancing. Identifying these connections can be non-trivial and will be discussed in Chapter 5.

Expression Definition 3

define ActualSource(F: FlowType, D: DFD) as set of CombinedLabels where CombinedLabels = ModifiedLabels | Labels such that
(Source(F) ? OffPage \implies ActualSource = { ProcessLabel(D) })
\land (Source(F) ? Label \implies ActualSource = { Source(F) })
\land (Source(F) ? ModifiedLabel \implies ActualSource = { Source(F) })
\land (Source(F) ? Vergence \implies ActualSource = \bigcup_{f \in VergSrc(F, D)} \{ ActualSource(f, D) \})

Expression Definition 4

define ActualDestination(F: FlowType, D: DFD) as set of CombinedLabels where CombinedLabels = ModifiedLabels | Labels such that
(Destination(F) ? OffPage \implies ActualDestination = { ProcessLabel(D) })
\land (Destination(F) ? Label \implies ActualDestination = { Destination(F) })
\land (Destination(F) ? ModifiedLabel \implies ActualDestination = { Destination(F) })
\land (Destination(F) ? Vergence \implies ActualDestination = \bigcup_{f \in VergDest(F, D)} \{ ActualDestination(f, D) \})

We have adopted a modified labeling scheme for stores and flows. This labeling scheme is used in Teamwork to resolve certain ambiguities that arise during flow

\(^5\) The invariant properties will assert that the DFD process label is the same as the label of the parent bubble which this DFD represents, thus insuring the modeling of a DFD hierarchy.
balancing [29]. In Chapter 5 we use modified labeling for a similar semantic purpose; however, we include its syntactic definition here rather than presenting it as an enhancement. Type Definition 10 defines the syntax of labels attached to stores and flows. A modified label $L$ is written "$<"modifier(L)"">"identifier(L)"$ when the modifier is a non-empty string and is otherwise written as $identifier(L)$. For example:

\[("new","word")\] is written \(<new>word\)
\[("\","wordlist")\] is written \(wordlist\)

Traditionally flows into or out of a store may be unlabeled and are intended to be implicitly labeled by the store label. Graphically we allow unlabeled flows into and out of a store, however, we implicitly use the store label as the flow label for unlabeled flows into and out of a store. Similarly, converging flows and divergent flows may be graphically unlabeled, but are implicitly labeled by the label attached to the associated convergent flow and diverging flow, respectively. Invariant Property 3 assures that store and flow labels are non-empty.

**Type Definition 10**

| ModifiedLabel | = 2-tuple( Modifier : string, Identifier : string); |

**Invariant Property 3**

(for any $L$ of type ModifiedLabel):

$Identifier(L) \neq "\$"

Each SA specification traditionally contains a context DFD which contains all the terminators, exactly one bubble, no stores, and is labeled with the model label
of the SA specification. All flows in the context DFD go from a terminator to the lone bubble or from the lone bubble to a terminator. Every terminator must be connected in some manner to the lone bubble. Invariant Property 4 formally defines this restriction.

**Invariant Property 4**

(for any \( M \) of type SA-Spec):

\[ \exists c \in \text{Processes}(M) \land c \notin \text{DFD} \land \text{ProcessLabel}(c) = \text{ModelLabel}(M) \]

\[ \land |Bubbles(c)| = 1 \land |\text{Stores}(c)| = 0 \]

\[ \land \forall f \in \text{Flows}(c) \Rightarrow \]

\[ \text{ActualSource}(f, c) \subseteq \text{Terminators}(M) \]

\[ \land \text{ActualDestination}(f, c) = \text{Bubbles}(c) \]

\[ \lor \text{ActualSource}(f, c) = \text{Bubbles}(c) \]

\[ \land \text{ActualDestination}(f, c) \subseteq \text{Terminators}(M) \]

\[ \land \left( \bigcup_{f \in \text{Flows}(c)} \text{ActualSource}(f, c) \bigcup \text{ActualDestination}(f, c) \right) \]

\[ = \text{Terminators}(M) \bigcup \text{Bubbles}(c) \]

Certain restrictions are traditionally enforced on the possible entities a flow can connect. A flow between two terminators represents communication outside the scope of the system and is not modeled. Note this restriction has already been specified by Invariant Property 4.

Stores are usually described as passive producers and consumers of data flows, thus we do not allow a flow between two stores. The manipulation of the contents of stores should be under the control of the system and in particular the portion (i.e., DFD) of the system in which the store appears. Thus a flow is not allowed between an off-page entity and a store. Note this also prevents flows between terminators and stores.

A self-looping flow, i.e., a flow whose source and destination is the same bubble,
represents communication at the next lower level in the DFD hierarchy and is traditionally not allowed. Typically such communication was modeled at the next lower level. In Chapter 7, we will present cases where self-looping flows provide a semantic convenience and thus do not disallow them here.

Finally, flow sources and destinations must be defined for the DFD in which the flows are contained and every store, bubble, and the DFD as a whole cannot be isolated from the rest of the SA-Spec, i.e., there must be an off-page connecting flow. These additional restrictions on flows are stated by Invariant Property 5.

**Invariant Property 5**

(for any \( M \) of type SA-Spec):

\[
\forall p \in \text{Processes}(M) \land p\notin \text{DFD} \land \text{ProcessLabel}(p) \neq \text{ModelLabel}(M) \Rightarrow \\
\forall f \in \text{Flows}(p) \Rightarrow \\
(\text{ActualSource}(f,p) \cap \text{Stores}(p) \neq \{\}) \\
\Rightarrow \text{ActualDestination}(f,p) \subseteq \text{Bubbles}(p)) \\
\land (\text{ActualDestination}(f,p) \cap \text{Stores}(p) \neq \{\}) \\
\Rightarrow \text{ActualSource}(f,p) \subseteq \text{Bubbles}(p)) \\
\land (\text{ProcessLabel}(p) \in \text{ActualSource}(f,p)) \\
\Rightarrow \text{ActualDestination}(f,p) \subseteq \text{Bubbles}(p)) \\
\land (\text{ProcessLabel}(p) \in \text{ActualDestination}(f,p)) \\
\Rightarrow \text{ActualSource}(f,p) \subseteq \text{Bubbles}(p)) \\
\land \text{ActualSource}(f,p) \cap \text{ActualDestination}(f,p) \subseteq \text{Bubbles}(p)) \\
\land \left( \bigcup_{f \in \text{Flows}(p)} \text{ActualSource}(f,p) \cup \text{ActualDestination}(f,p) \right) \\
= \text{Stores}(p) \cup \text{Bubbles}(p) \cup \{ \text{ProcessLabel}(p) \}]
\]

Because we have stated these restrictions in the context of actual sources and actual destinations, we may have been more restrictive then necessary in the presence of converging and diverging flows. However, this is difficult to evaluate since we have not presented an interpretation of the meaning of converging and diverging flows. For
example, consider Figure 4.4. Since the flow labeled Z has a store as both an actual source and an actual destination, it would not be allowed. However, if the interpretation is that X flows from Store1 to Bubble2 and Y flows from Bubble1 to Store2 and nothing really flows from Store1 to Store2 then we have excessively restricted our domain to exclude this example. In Chapter 5 we do develop an interpretation of converging and diverging flows that makes Invariant Property 5 excessively restrictive in certain cases (i.e., Figure 4.4 would not be allowed) depending on the labels of flows that converge and diverge. This excessive restriction does however enforce guidelines for avoiding unnecessary use of converging and diverging flows. Thus converging and diverging flows should only be used to help simplify a DFD, not make it more complex.

![Diagram](image)

**Figure 4.4: Excessive Flow Restriction**

Recall that ProcessType is declared as an alternative of DFD and P-Spec. To represent the DFD hierarchy, we associate with each DFD bubble a ProcessType object. If a DFD bubble is associated with a DFD object then that bubble is decomposed into a DFD. For example, Bubble0 and Bubble3 in the example SA specification in
Figure 4.1 are associated with DFDs—Bubble0 decomposed into DFD0 and Bubble3 decomposed into DFD3. Each of those DFDs' bubbles must also be associated with a ProcessType object.

If a DFD bubble is associated with a P-Spec object then that bubble represents a bubble at the bottom of the DFD hierarchy. For example; Bubble1, Bubble2, Bubble3.1, Bubble3.2, and Bubble3.3 in the example SA specification in Figure 4.1 are all associated with a P-Spec—Bubble1 decomposed into P-Spec1, Bubble2 decomposed into P-Spec2, etc. However, not all bubbles in any given DFD must be associated with the same kind of ProcessType. Some may be associated with a DFD and some others with P-Specs. For example, Bubble1 and Bubble3 in DFD0 of the example SA specification in Figure 4.1 do not decompose into the same kind of ProcessType object.

A P-Spec specifies a primitive process and is a leaf-like component of the tree-like hierarchy of DFDs. Other than the process label and the labels of in-flows and out-flows associated with a P-Spec, the traditional methods of specifying a P-Spec are usually informal, relying on English or pseudo-code descriptions of functionality. Type Definition 11 defines this limited formalism. The in-flows and out-flows are labeled analogously to flows in a DFD, but are also used as formal parameters to refer to specific values in the body (i.e., pseudo-code) of a P-Spec. In Type Definition 11 the InFlows and OutFlows could be defined as sets of ModifiedLabels, however, defining them as sets of FlowTypes facilitates the expression of particular invariant properties. P-Spec's are typically represented textually, not graphically; in-flows are labeled as data_in parameters and out-flows are labeled as data_out parameters (Figure 4.5). Invariant Property 6 defines the source and destination of an in-flow as
off-page and the process label, respectively, and defines the source and destination of an out-flow in a reverse manner. This will set the stage for the balancing of flows between DFD bubbles and P-Specs. In Chapter 7 we question the exact nature of primitive processes and extend our abstract model to include formal specifications of a generalized notion of primitive processes. In Example 4 on page 78 we complete the set of processes in the example by filling in the details of P-Specs.

<table>
<thead>
<tr>
<th>P-Spec3.2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>data_in:</td>
<td>Flow2</td>
</tr>
<tr>
<td>data_in:</td>
<td>Flow9</td>
</tr>
<tr>
<td>data_out:</td>
<td>Flow7.2</td>
</tr>
<tr>
<td>data_out:</td>
<td>Flow10</td>
</tr>
<tr>
<td>Body:</td>
<td>&quot;pseudo-code goes here&quot;</td>
</tr>
</tbody>
</table>

Figure 4.5: Example P-Spec

**Type Definition 11**

\[
P\text{-Spec} = 4\text{-tuple}( \text{ProcessLabel} : \text{Label}, \\
\text{InFlows} : \text{set of FlowType}, \\
\text{OutFlows} : \text{set of FlowType}, \\
\text{Body}: \text{string});
\]

**Invariant Property 6**

(for any \( P \) of type P-Spec):

\[
\forall f \in \text{InFlows}(P) \implies \text{Source}(f) \in \text{OffPage} \land \text{Destination}(f) = \text{ProcessLabel}(P) \]

\[
\land \forall f \in \text{OutFlows}(P) \\
\implies \text{Source}(f) = \text{ProcessLabel}(P) \land \text{Destination}(f) \in \text{OffPage}
\]

Recall, the set of processes defined by an SA-Spec can be viewed as a tree-like hierarchy. The context diagram forms the root of the tree and has one child which
Example 4

(ModelName,
   {Terminator1, Terminator2, Terminator3},
   {}
)

{(ModelName,
   {Bubble0},
   {Flow1, Terminator1, Bubble0, Flow2, Terminator2, Bubble0, Flow3, Bubble0, Terminator3}),

(DFD0,
   {Bubble1, Bubble2, Bubble3},
   {Store1},
   {(Flow1.1, Null, Bubble1), (Flow1.2, Null, Bubble1), (Flow2, Null, Bubble3),
    (Flow3, Bubble2, Null), (Flow4, Bubble1, Store1), (Flow5, Store1, Bubble2),
    (Flow6, Bubble1, Bubble3), (Flow7, Bubble1, 0), (Flow7, Bubble3, 0),
    (Flow7.0, Bubble2))},

(DFD3,
   {Bubble3.1, Bubble3.2, Bubble3.3},
   {},
   {(Flow2, Null, Bubble3.2), (Flow6, Null, Bubble3.1), (Flow7.1, Bubble3.3, Null),
    (Flow7.2, Bubble3.2, Null), (Flow8, Bubble3.1, Bubble3.3), (Flow9, Bubble3.1, Bubble3.2),
    (Flow10, Bubble3.2, Bubble3.3)}),

(P-Spec1,
   {{(Flow1.1, Null, P-Spec1), (Flow1.2, Null, P-Spec1)}
    {{(Flow4, P-Spec1, Null), (Flow6, P-Spec1, Null), (Flow7, P-Spec1, Null)}},

(P-Spec2,
   {{(Flow5, Null, P-Spec2), (Flow7, Null, P-Spec2)}
    {{(Flow3, P-Spec2, Null)}},

(P-Spec3.1,
   {{(Flow6, Null, P-Spec3.1)}
    {{(Flow8, P-Spec3.1, Null), (Flow9, P-Spec3.1, Null)}},

(P-Spec3.2,
   {{(Flow2, Null, P-Spec3.2), (Flow9, Null, P-Spec3.2)}
    {{(Flow7.2, P-Spec3.2, Null), (Flow10, P-Spec3.2, Null)}},

(P-Spec3.3,
   {{(Flow8, Null, P-Spec3.3), (Flow10, Null, P-Spec3.3)}
    {{(Flow7.1, P-Spec3.3, Null)}},

{(Flow1, . . .), (Flow1.1, . . .), (Flow1.2, . . .), (Flow2, . . .), (Flow3, . . .),
 (Flow4, . . .), (Flow5, . . .), (Flow6, . . .), (Flow7, . . .), (Flow7.1, . . .),
 (Flow7.2, . . .), (Flow8, . . .), (Flow9, . . .), (Flow10, . . .), (Store1, . . .)}

}
is the process associated with the single bubble contained in the context diagram. Each internal node in the tree is a DFD process associated with a bubble in the DFD of its parent node. The leaves of the tree are the P-Specs. The existence of this hierarchy is asserted by Invariant Property 7. We require that each process, either DFD or P-Spec, be uniquely determined by its process label. Furthermore, we require that each process label correspond to either a bubble in some other DFD or the model label. This prevents the existence of processes that do not contribute to the hierarchy of DFDs and avoids direct circularities in the DFD hierarchy. In order to prevent the existence of indirect circularities in the DFD hierarchy, we require that any particular bubble label appear in only one DFD. Similarly we require that any store label appear in only one DFD. This requirement localizes the references made to that store since any flow into or out of a store must be connected to a bubble in the same DFD. Invariant Property 7 characterizes these restrictions.

**Invariant Property 7**

(for any \( M \) of type SA-Spec):

\[
| \text{Processes}(M) | = \left| \{ \text{ProcessLabel}(p) \mid p \in \text{Processes}(M) \} \right|
\]

\[\land \forall p [ p \in \text{Processes}(M) \Rightarrow \text{ProcessLabel}(p) = \text{ModelLabel}(M) \]

\[
\lor \exists d [ d \in \text{Processes}(M) \land d \neq p \land \exists \text{DFD} \land \text{ProcessLabel}(p) \in \text{Bubbles}(d) ]
\]

\[\land \forall d_1, d_2 [ d_1 \in \text{Processes}(M) \land d_1 \neq d_2 \land \exists \text{DFD} \land d_2 \in \text{Processes}(M) \land d_2 \neq \text{DFD} \]

\[\land d_1 \neq d_2 \Rightarrow \text{Bubbles}(d_1) \cap \text{Bubbles}(d_2) = \{ \}
\]

\[\land \text{Stores}(d_1) \cap \text{Stores}(d_2) = \{ \}
\]

In our example SA specification, these restrictions would require the following string equalities to hold:

- Bubble0 = DFD0, Bubble1 = P-Spec1, Bubble2 = P-Spec2, Bubble3 = DFD3,

- Bubble3.1 = P-Spec3.1, Bubble3.2 = P-Spec3.2, and Bubble3.3 = P-Spec3.3
The numbering scheme we used for the labels of processes is a commonly used CASE tool technique for associating bubbles with processes. Although not strictly required, the numbers do make the matching of processes with bubbles easier. Both DFD and P-Spec tuples could be augmented with an additional field for the process number.

4.2.2 The DD

A DDE traditionally defines the composition of the data values associated with a flow or store label. Recall that a store or flow label is composed of two components; a modifier and an identifier. The Teamwork tool associates a DDE with each unique identifier component of a flow or store label. We could create a separate DDE for each uniquely labeled flow or store including the use of the modifier component. However, in Chapter 5, we develop an interpretation of the data composition associated with a flow or store that is modifier independent. Thus all flow labels with the same identifier component would be described as having the same composition. This leads to three possible approaches for defining flow and store labels in the DD:

1. Include in the DD an entry for each uniquely labeled flow or store including the use of the modifier. The set of DDEs associated with flows labeled with the same identifier component would each describe the same composition of data values. Each DDE could also include an additional attribute that distinguishes the particular modifier used.

2. Include in the DD an entry for each uniquely labeled flow or store including the use of the modifier as above. (Call these modified DDEs.) Each modified

---

6The definition and interpretation of what is meant by *composition of the data values* is another issue dealt with in Chapter 5.
DDE could then reference an additional DDE that describes the composition of data values. (Call these unmodified DDEs.) This places the description of the composition associated with a particular label identifier in a central location making changes to the composition easier to manage. Each modified DDE could still include additional attributes that distinguishes the particular modifier used.

3. Include a single DDE for each unique identifier component of a modified label. This single DDE would include the description of the data composition and could also include distinguishing attributes associated with each particular modifier. This approach decreases the size of the DD and places the composition description in a central location. This approach can be seen as an analogy to the typical English dictionary, where each word has several definitions and the correct definition depends on the context. In the case of flow and store labels, the particular modifier used distinguishes the context. For example, suppose in an SA specification of a spelling checker we identified three different flows whose composition described a list of words. These flows might represent the words from the file to be checked, words correctly spelled in the on-line dictionary, and words from the file that did not appear in the dictionary. Suppose we labeled these flows \(<file>wordlist\), \(<dictionary>wordlist\), and \(<unmatched>wordlist\), respectively. The DDE for wordlist might describe a wordlist as a sequence of words and furthermore might state that every word in \(<unmatched>wordlist\) must also be in \(<file>wordlist\) and may not be in \(<dictionary>wordlist\).
We adopt the last approach and associate a DDE with the identifier component of a label attached to a flow or store. Type Definition 12 defines a DDE.

**Type Definition 12**

\[
\text{DDE} = 2\text{-tuple} \left( \text{EntryName} : \text{string}, \text{EntryDef} : \text{EntryType} \right);
\]

Traditionally, the definition of a DDE is either provided by a composition expression or flagged as elementary. This alternative is provided by Type Definition 13. Elementary DDEs usually include a comment describing a range of potential values, but there are no formal requirements for comments. Traditionally an elementary DDE's name should be chosen to be self-descriptive and is said to thus not require a composition definition. This is characterized by Type Definition 14.

**Type Definition 13**

\[
\text{EntryType} = \text{Composition} \mid \text{Elementary};
\]

**Type Definition 14**

\[
\text{Elementary} = 0\text{-tuple}( );
\]

The composition associated with a non-elementary DDE is defined by a composition expression which apparently denotes the potential values associated with a flow or store. Two approaches can be taken when interpreting the description of the potential values of a flow or store. One method is to consider the composition expression as describing the type structure of the potential values. Using this characterization, a composition expression can be viewed as defining a domain of potential
values similar to the domain descriptions possible with SPECS. The second method is to consider the composition expression as a lexical description of the potential values. In Chapters 5 and 6, we examine the roles composition expressions fulfill in SA specifications. Since a composition expression itself is a lexical object (i.e., it has a particular allowable syntax) we do not provide an abstract model type description of its syntax, but rather rely on a BNF description of its syntax. Type Definition 15 refers to a BNF describing the language of composition expressions. Table 4.1 provides the BNF description of this language and a brief definition of the lexical interpretation of composition expressions. We assume the usual BNF definition of identifier.

Type Definition 15
Composition = BNF[CompExpr];

Every flow label and store label must be represented by an entry in the DD. Also every subordinate modified label that appears as part of a composition expression must also be defined by a DDE. Note that the fifth alternative definition of CompExpr in Table 4.1 is a modified label. This restriction is characterized by Invariant Property 8. We assume the existence of a parser for composition expressions that allows us to construct the set of all modified labels that appear in any composition expression. We use Expression Definition 5 to refer to this set of all modified labels that appear in any particular composition expression.\

\footnote{Expression Definition 5 is informal stated.}
Table 4.1: DDE Composition Expression Syntax

```
CompExpr ::= CompExpr { '+' CompExpr }+
  | [ ' ' CompExpr { ' ' | ' ' CompExpr '+' ' ' } ]
  | [ pos-integer ] { ' ' CompExpr ' ' [ pos-integer ]
  | '(' CompExpr ')' |
  | [ modifier ] identifier
  | literal
modifier ::= '<<identifier>>'
literal ::= "" text ""
text ::= any string that does not contain "
pos-integer ::= any positive integer

+    means concatenation
[ 1 ] means alternation
{}  means iteration (bounds default to 0 and oo)
()  means optional, i.e., (X) = {X}1
```

Invariant Property 8

(for any $M$ of type SA-Spec):

\[
\{ Identifier(y) \mid \exists p \in Processes(M) \land p?DFD \land y \in Stores(p) \}\}
\cup \{ Identifier(FlowLabel(z)) \mid \exists p \in Processes(M) \land p?DFD \land z \in Flows(p) \}\}
\cup \{ Identifier(FlowLabel(v)) \mid \exists p \in Processes(M) \land p?P-Spec
\land v \in InFlows(p) \cup OutFlows(p) \}\}
\cup \{ EntryName(w) \mid w \in DataDictionary(M) \} =
\{ Identifier(x) \mid \exists d \in DataDictionary(M) \land EntryDef(d)?Composition
\land x \in ExpLabels(EntryDef(d)) \}\}

Expression Definition 5

define $ExpLabels(C:Composition)$ as set of ModifiedLabel
such that

$ExpLabels = \{ x \mid x$ was found in a parse of $C$ \}
4.2.3 DFDs and DDEs Combined

We require that labels associated with terminators, processes, and DDEs all form disjoint sets as given by Invariant Property 9. This avoids confusion when referring to a particular DFD component by name.

Invariant Property 9

(for any \( M \) of type SA-Spec):
\[
\{ \text{ProcessLabel}(p) \mid p \in \text{Processes}(M) \} \cap \text{Terminators}(M) = \{ \}
\]
\[
\wedge \{ \text{EntryName}(y) \mid y \in \text{DataDictionary}(M) \} \cap \text{Terminators}(M) = \{ \}
\]
\[
\wedge \{ \text{ProcessLabel}(p) \mid p \in \text{Processes}(M) \}
\cap \{ \text{EntryName}(y) \mid y \in \text{DataDictionary}(M) \} = \{ \}
\]

Previously in Invariant Property 7 we required a SA-Spec to include the complete hierarchy of DFDs and P-Specs. This hierarchy was based on the decomposition of a DFD bubble into either a DFD or a P-Spec. This decomposition of a DFD bubble into either a DFD or P-Spec must also include a balancing of the parent bubble’s inflows and out-flows with the off-page flows of the child process. However, the off-page flows in a child process may appear in a decomposed form. De Marco describes this as a parallel decomposition of both function and data [18]. The balancing of the parent flows with the child flows is described in terms of the DDE composition expressions associated with flow labels. Thus the allowable flow decomposition that can occur between a parent bubble and a child process is defined by the DDE associated with the flows involved.

A convergent or diverging flow must also balance with its associated converging or divergent flows. Allowable forms of vergence are also based on the DDE composition expressions associated with the flows involved in the vergence. Similarly flows
into and out of a store must be labeled by either the store label or a subordinate of
the store as defined by the DDE composition expression associated with the store
label.

Stating these restrictions as an Invariant Property is somewhat problematic. The
classic references by De Marco [18] and Gane and Sarson [22] as well as the more
current reference by Yourdon [66] do not provide a clear definition of flow balancing.
The manuals of Teamwork provide an algorithmic description of flow balancing for
process decomposition but are unclear about stores [29]. Additionally, this algorithmic
description does not differentiate between the various types of operators (i.e.,
catenation, alternation, etc.) used to construct a DDE composition expression.
Other references [27, 59] provide graphical representations of allowable labelings of
convergence and divergence flows, but these references do not relate these allowable
labelings to particular DDE composition expressions.

We do not attempt to describe the flow balancing restriction by an invariant
property. In Chapter 5 we discuss alternative interpretations of flows, flow labels,
and DDE composition expressions that will serve to provide a definition of flow
balancing. We begin by considering what it means for a vergence to balance and
extend these considerations to the balancing of flows in DFD process decomposition.
These considerations are dependent upon providing particular semantics for the DDE
composition operators. In Chapter 8 we consider interpretations for flow balancing
of stores.

Each of the following four chapters builds on and modifies the syntactic definition
of an SA specification. Our goal is to enhance the definition of an SA specification to
provide a formal framework which will allow for the rigorous and precise interpreta-
tion of the meaning of an SA specification. We do this by enhancing those components of an SA specification that are syntactically ill-defined (e.g., P-Specs) and by providing semantic interpretations of the syntactic components of an SA specification, especially those components whose current interpretations are either inadequate or non-existent. From these formalized definitions and interpretations of SA components, we develop in Chapter 9 an operational semantics for an SA specification as a whole.
CHAPTER 5. UNDERSTANDING FLOW BALANCING

5.1 Flow Balancing

The main goal of this chapter is to provide a definition of flow balancing. Every SA specification in the domain of completed SA specifications must satisfy these flow balancing constraints. Every vergence point must balance with respect to the verging and vergent flows and every child processes’ off-page flows must balance with respect to the in-flows and out-flows of its parent bubble. To form an interpretation of what makes sense for flow balancing we need to understand exactly what a flow represents. We do this in three parts:

1. Flow Labels: In Section 5.2 we consider the flow label. Other than naming a directed edge in a DFD, what does a flow label label? The interpretation of the role flow labeling plays must also include the use of flow labels for labeling the data_in and data_out parameters in a P-Spec. The particular interpretation chosen is really an expressiveness choice. How do we view flows in an SA specification and how can we interpret them while still facilitating manageability of SA specifications?

2. Flow Value Behavior: Central to the interpretation of flow balancing is the representation of data or value instances in an SA specification. An SA speci-
ification specifies a dynamic system; data is created, used, and discarded as a function of time. Yet often the traditional SA approach produces a static view of the system. Traditional SA specifications simply are not operationally interpreted. Flows and stores are static representations of this dynamically existing data. We need to understand how a static flow represents this dynamic behavior of data. In Section 5.3 we consider the data instances associated with flows. We call these flow values. Several flow value behaviors are suggested by various authors for characterizing the dynamic behavior of data instances. However, the choice of a particular behavior for each flow affects how we interpret other parts of the specification. In particular, how do the various other components of DFDs, bubbles, stores, and terminators interact via the connecting flows. Formal interpretation of the potential behavior of flow values is important for a clear understanding of how data moves through the system.

3. Flow Abstraction: The decomposition of a DFD bubble into another DFD or a P-Spec is a representation of process specialization. The parent bubble is an abstraction of the underlying DFD. A similar form of abstraction is possible for flows. We term the representation of several flows by a single flow flow abstraction. Flow abstraction occurs in two forms. One is the convergence and divergence of flows in a single DFD which we call vergence flow abstraction. Thus a convergent flow is really a flow abstraction of the associated converging flows. The other form of flow abstraction occurs in parallel with process abstraction. A single in-flow or out-flow of a parent bubble may actually be represented by several off-page flows in the associated child process. The off-page flows in a child process that balance with a particular flow into or out
of the parent bubble have been abstracted at the higher level. We call this *decomposition flow abstraction*.

Traditionally the allowable syntactic occurrences of flow abstraction depend on the DDE composition expressions associated with the flow labels. For example, if a flow labeled Z diverged into two flows labeled X and Y, respectively, then Z needs to be defined in terms of X and Y in the DD. One possibility is that Z = X + Y. However, to form an operational interpretation of an SA specification, flow abstraction may also need to consider the particular flow value behaviors of the flows involved. In Section 5.4 we consider several possible interpretations of flow abstraction and suggest an interpretation that provides both expressive utility and manageable simplicity. However, while a formal interpretation of flow abstraction is crucial for understanding how data moves through the specified system, and thus for forming an operational interpretation, the particular interpretation chosen is not as important as making a choice, because a choice resolves the ambiguity present.

Abstraction plays a major role in an SA specification. Process abstraction allows us to view large portions of the system as a single unit at higher levels in the DFD hierarchy. In the absence of process abstraction (i.e., if we flattened the hierarchy of DFDs) an SA specification describes a single "football field sized" DFD which contains all the terminators, all the stores, and only bubbles which are abstractions of P-Specs. For example, Figure 5.1 provides the "football field sized" DFD for the hierarchy of DFDs given in Figure 4.1. Constructing this "football field sized" DFD requires a definition of flow balancing. Understanding this "football field sized" DFD requires an interpretation of flow abstraction. The role of flow balancing is to assure
that the “football field sized” DFD exists. Our goal is to form an interpretation of flow abstraction that can be used to define flow balancing and make sense of the “football field sized” DFD. However, we want this interpretation to be useful for the specification of systems and also to be as consistent as possible with the usual informal semantics of traditional SA specifications.

Figure 5.1: “Football Field Sized” DFD

5.2 Flow Labeling

Traditionally a flow is said to represent data in motion [18]. A flow appears as a labeled directed edge in a DFD and is specified by a DDE associated with the label attached to the flow in the DFD.
Many authors spend a great deal of effort describing how to choose appropriate labels for flows, e.g., choose nouns not verbs, don't be "wishy-washy", etc., [18, 22, 66]. The basic premise is that a flow label names an object and thus the label should be a descriptive noun that describes that object. In fact, traditional P-Spec specifications use flow labels as formal parameters, i.e., values, in pseudo-code descriptions of P-Spec functionality. However, De Marco characterizes a flow as a "pipeline through which packets of information of a known composition flow" [18]. De Marco's definition of a flow implies that a flow is more like a type—it has structure. This structure is described by the composition expression associated with the DDE of the flow label. As mentioned previously, a composition expression can be interpreted as either a lexical description of value or as a type structure description of a domain. There seems to be some confusion then as to whether a flow label names a value or a type.

This confusion is really an expressibility issue. How often have you seen an Ada-like procedure whose formal parameter Foo is declared to be of type FooType? So should we label our flows with value names or type names? The goal is to ensure clear communication between the users of SA specifications. Thus it's not so much that one method is correct and the other incorrect, but rather the issue exists and the choice needs to be made. Can we then decide which method is preferable? What are the advantages and disadvantages of using each of the approaches? Consider a simple spelling checker system described as follows:

The user provides the name of a document file which is to be checked for spelling mistakes. The spelling checker system will check each word in the document file to see if it is in the on-line dictionary. The users will
have the option of adding words not found to the dictionary. Words not found in nor added to the dictionary are output to a misspelled word file.

5.2.1 Value Named Flows

Figure 5.2 provides a context diagram for an SA specification of the spelling checker system using value named flows. The User provides the document-name. The document is retrieved from the FileSystem by document-name. The Dictionary provides good-words that are correctly spelled. The User is sent unmatched-words and provides dispositions to either add new-words to the Dictionary, or bad-words to the MissSpelledWords.

Figure 5.2: Context Diagram Using Value Naming

Figure 5.3 is a decomposition of the Spelling Checker context diagram bubble.
Form_Wordlist retrieves the document and sends the check-words to Match. Match compares check-words with good-words and sends the unmatched-words to both the User and Handle_Unmatched_Words. The dispositions provided by the User are used by Handle_Unmatched_Words to decide if a word belongs to bad-words or new-words. Figure 5.4 gives a pseudo-coded P-Spec for the Match bubble. Notice how the flow names are used as formal parameters. Consider the advantages and disadvantages of using the value naming method for flows.

![Figure 5.3: Decomposition Using Value Naming](image)

Advantages:

- Value naming allows identification of flows carrying the same value. Although many authors require that every flow be uniquely labeled, relaxing
TITLE: Match

INPUT/OUTPUT:
check-words : data.in
good-words : data.in
unmatched-words : data.out

BODY:
foreach word in check-words do
  if word is not in good-words
  then add word to unmatched-words

Figure 5.4: P-Spec Using Value Naming

this requirement allows us this identification. Consider the flows labeled document-name in both Figure 5.2 and 5.3. The value that flows into Spelling_Checker_System and thus Form_Wordlist is exactly the same as the value that flows out. The two flows are distinguished uniquely by their direction of flow.

The two flows labeled unmatched-words in Figure 5.3 also carry the same value. However, this case represents a different situation than the flows labeled document-name. Both flows labeled unmatched-words appearing in Figure 5.3 balance with the single data_out flow unmatched-words in P-Spec Match in Figure 5.4. In Section 5.4 we will see that the values flowing represent copies of the same value produced by Match. This could also be represented by a single out-flow from the Match bubble labeled unmatched-words that diverged into two unlabeled flows.
• Value naming also facilitates the writing of P-Specs. The flow name can be used to refer to the value in the P-Spec body as shown by P-Spec Match in Figure 5.4.

Disadvantages:

• Value naming creates a large number of redundant DDEs. For example consider the flows labeled bad-words, check-words, good-words, new-words, and unmatched-words. Each of these flows might be characterized as an iteration of words.

• Because of this redundancy the size of the data dictionary (DD) grows quickly and becomes intellectually less manageable.

5.2.2 Type Named Flows

As an alternative, consider Figure 5.5 in which the value naming flow labels have been replaced by type naming flow labels. The label word-list is used for flows that represent an iteration of words. Flow labels file-id and file represent type structures for a string and an iteration of characters, respectively. (Note that even though a string is a sequence of characters, we usually think of the name of a file as a whole, i.e., string and the contents of a (text) file in terms of individual characters.) The flow label decisions represents an iteration of individual decisions as to the correctness of words not found in the Dictionary. Consider the advantages and disadvantages of using the type naming method for flows.

Advantages:
Figure 5.5: Decomposition Using Type Naming
• Type naming results in few, if any, redundant DDEs. It certainly is still possible to have multiple DDEs which define the same type structure, however, it is not necessary to describe a particular type structure in more than one DDE. There is also the possibility that one DDE is defined as an alias for another DDE.

• The DD is smaller and thus more manageable.

Disadvantage:

• Type naming causes a flow distinction problem. Flow sources and destinations cannot distinguish between two distinct flows with the same label and direction which are intended to carry different values from the same domain. This is especially problematic in referring to values in a P-Spec body. For example, how do we distinguish the two distinct parent bubble in-flows labeled word-list in the P-Spec for Match? Since we have defined the flows in a P-Spec as a set (Type Definition 11), we actually cannot distinguish two like labeled in-flows or out-flows in a P-Spec.

5.2.3 Modified Type Named Flows

To overcome the disadvantages of both value naming and type naming we use the modified labels whose syntax was presented in Chapter 4. First we consider the Identifier component of the modified label to be a type name. To distinguish between two flows with the same type structure we attach a modifier to the flow label. Although the modifier is not required unless the flow distinction problem arises, we do not preclude its use as a mnemonic aid for readability and clarity. Figure 5.6 shows
the decomposition of the Spelling_Checker_System using modified type naming. In a P-Spec body we can use the modifier to refer to distinct values. Figure 5.7 shows the P-Spec for Match using modified labels.

Interpreting the labels attached to flows as either value names or type names addresses an expressibility issue. The solution we presented is only one possibility. There are solutions other than the modified type naming solution we have suggested. The real problem is that such issues exist and need to be addressed to facilitate effective communication of ideas. If one person is reading flow labels as types and another as values, then communication is breaking down.

How does flow naming effect flow balancing? If the identifier field of a flow label is being interpreted as a type name and is defined in the DD by a composition expression

Figure 5.6: Decomposition Using Modified Type Naming
then it seems we are suggesting that composition expressions be interpreted as type structure definitions. Since flow balancing is traditionally defined in terms of the composition expressions, it would seem that our flow abstractions will be based on type structures and balancing represents composition and decomposition of types. However, this is not exactly the approach taken. In Chapter 6 we replace the DDE composition expression's used for describing type structure with SPECS ADTs. In Section 5.4 we use composition expressions to define allowable flow abstractions, but we use a flow value behavior approach to interpret flow abstractions. In the next section we identify several different kinds of flow values based on the behavior of these values over time.
5.3 Flow Value Behavior

In this section we address the existence and behavior of instances of data represented by flows in a DFD. We call these instances flow values. What is the relationship between a flow and the data it represents? Does data occur in a one-to-one relationship with a flow, i.e., does each flow represent a single flow value at any particular time? Can this flow value change over time? If so, how does it change?

Since we have not yet provided an interpretation of flow abstraction, we approach these questions from a perspective in which flow abstraction is not an issue. For the remainder of this section assume that convergence and divergence of flows are not allowed. Furthermore, assume that flows into and out of a parent DFD bubble balance in a one-to-one relationship with the off-page flows in the child process; i.e., for every in-flow and out-flow to a parent bubble exactly one off-page in-flow or off-page out-flow (with the same exact label) appears in the child process. In effect, we are considering the flows as they might appear in a “football field sized” DFD, which does not contain vergences. By requiring flows to balance in this one-to-one relationship, we avoid the issue of putting together or taking apart flow values that combine or split in a vergence or decomposition flow abstraction.

Several authors, Ward [59, 12], Hatley and Pirbhai [27], De Marco [18], Gane and Sarson [22], used the terms continuous and discrete to describe data in an SA specification. In some cases these terms are applied to flows in DFDs, in other cases they are applied to DDEs. In the remainder of this section we describe our interpretations of how each of these authors are using these terms and from these interpretations we develop a description of three possible flow value behaviors.

We characterize our interpretations of the terms continuous and discrete as either
describing the existence of data values, the change in data values, or the domain of data values. The later interpretation will not contribute to our description of flow value behavior. A combination of the data existence and data change interpretations is used to describe three possible flow value behaviors. Each of these three flow value behaviors has an analogous implementation level software or hardware representation. Although we encourage the practicing software engineer to use these analogies to help understand SA specifications, we do not intend that these flow value behaviors define a direct mapping to an implementation or design. There may be heuristics which, based on flow value behaviors as well as other factors, can help guide the choice of a design representation or even implementation platform. However, such considerations are not included in this thesis.

5.3.1 Continuous versus Discrete Existence

Ward uses discrete and continuous to categorize flows by their relationship to the flow values on the flows [59, 12]. Ward divides flows into continuous flows and discrete flows. Ward's use of continuous and discrete provides an interpretation of flow value existence over time. Ward describes a transformation schema which is essentially a token based execution of a DFD, including the use of control flows and control bubbles. Recall from Chapter 2, control flows and control bubbles are part of an extension of the original SA specification technique for the specification of real-time systems. Ward terms his resulting SA specification an implementation model and rightly so, as the control flows and control bubbles infuse an implementation strategy for controlling the data transformation bubbles. A token on a flow represents an individual instance of a flow value. Ward's use of continuous and discrete to categorize
flows is based on the description of how tokens are placed on and removed from the flows. Ward's categorization is characterized as follows:

**Continuous Flow:** A continuous flow whose source is a terminator always carries a token. A continuous flow whose source is a data transformation bubble carries a token whenever the data transformation bubble carries a token and also during output delays. A data transformation bubble carries a token when the controlling bubble has given it permission to execute. The data transformation token disappears during an output delay, which occurs when the data transformation bubble has accepted inputs and is required to wait before producing new outputs.

**Discrete Flow:** The placement of a token on a discrete flow indicates an instance of the flow, i.e., an occurrence of an associated value, has occurred at that point in time. At most one discrete flow is allowed as input to a data transformation. When a token appears on an input discrete flow, the data transformation executes and removes the token from the discrete flow.

We term this interpretation of continuous and discrete continuous-existence and discrete-existence. A continuous flow token exists continuously from the perspective of the bubble that uses that continuous flow as input. Whenever the bubble is able to execute and is not currently executing, the token and thus flow value is accessible on the continuous flow. A discrete flow token is placed on a discrete flow by the producing bubble and then removed from the flow by the receiving bubble. The token and thus flow value exist for only a discrete period of time.
Assuming data transformation bubbles always have permission to execute, which will be the case in Chapter 9, Ward's interpretation of a continuous flow is analogous to a shared variable in a concurrent system. The flow value is always there. The source of the flow serves as the writer of the shared variable and the destination serves as the reader. Recall this representation as a shared variable is only an analogy. If this flow were to be implemented as a shared variable, then and only then would we need to worry about mutual exclusion and associated deadlock issues.

Ward's interpretation of a discrete flow is analogous to a fixed message channel in a distributed system. Message passing is a more general communication mechanism than shared variables and supports development of distributed systems [16]. The source of the flow serves as the message sender. The destination of the flow serves as a message receiver. The placement of a token on a discrete flow by the source represents the act of sending a message. The removal of the token by the destination represents the act of receiving the message. Ward does allow the existence of multiple tokens on a discrete flow. We extend the analogy to include the existence of multiple messages on a message channel. However, since a source can only send one message at a time, we interpret a discrete flow as a FIFO ordered queue of messages.

5.3.2 Continuous versus Discrete Change

Hatley and Pirbhai also divide flows into continuous flows and discrete flows [27]. However, Hatley and Pirbhai interpret continuous and discrete as representing the way a value on a flow changed over time. The value associated with a continuous flow defined a continuous function with respect to time as in Figure 5.8. A continuous flow would necessarily require a continuous domain of potential values from which to
form this function. Ideally, the only continuous domain possible is some subset of the reals. However, if an integer valued flow has a current value of \( n \) and we know that the next value will always be either \( n - 1 \) or \( n + 1 \) then we might consider the flow as defining a continuous function with respect to time. This flow is characterized as continuous because the next value to appear on the flow was either one greater or one less than the previous value.

Hatley and Pirbhai's discrete flow value defined a discrete function with respect to time as in Figure 5.9. An example might be a flow whose value is a customer's telephone number. The next customer's telephone number in general will not numerically precede or follow the current customer's telephone number. At some discrete time point the value on the flow instantaneously changes to the next value. We term this interpretation of continuous and discrete as continuous-change and discrete-change.

![Figure 5.8: The Value of a Continuous Flow](image_url)
5.3.3 Continuous versus Discrete Domain

De Marco also used the terms discrete and continuous for describing data values [18]. However, De Marco referred to the DDE composition expression's description of values. Elementary DDEs and DDEs defined only in terms of literals are described as being either continuous or discrete. For example, a composition expression which was an exclusive-or of literal values (e.g. ["red" | "blue" | "green"])) defines a discrete DDE. This use of continuous and discrete refer to the domain of values a particular DDE could represent. Although Hatley's use of continuous implies a continuous domain, De Marco's use of continuous and discrete does not refer directly to particular flows. For example, De Marco does not exclude the possibility that the DDE associated with a particular flow is defined as a concatenation of a discrete DDE and a continuous DDE, e.g., $Z = X + Y$, where $X$ is defined by a continuous DDE and $Y$ is defined by a discrete DDE. However, the domain of potential values a flow may represent does not directly describe the behavior of these values over time.
5.3.4 Interpreting Flow Value Behaviors

In Table 5.1 we consider the cross product of the two interpretations—existence and change—for continuous and discrete flows.

Table 5.1: Flow Value Change versus Flow Value Existence

<table>
<thead>
<tr>
<th></th>
<th>continuous-existence</th>
<th>discrete-existence</th>
</tr>
</thead>
<tbody>
<tr>
<td>continuous-change</td>
<td>analog</td>
<td></td>
</tr>
<tr>
<td>discrete-change</td>
<td>persistent</td>
<td>consumable</td>
</tr>
</tbody>
</table>

A continuously changing value defined over a continuous domain is behaving much like an analog signal. If we are only interested in the current value of the analog signal then the old value can be thrown away as it is replaced by a new value. For this reason, we characterize a continuously changing, continuously existing flow as an *analog* flow.

Analog signals usually change in minute increments in a short period of time. If the old values of the analog signal are queued up on a discrete-existence flow into another process, then that process must execute repeatedly and rapidly to avoid a backlog of old values. In fact, the receiving process must execute on the average as fast as the analog value changes. Such behavior occurs almost exclusively in hardware and in this case the queue of values could then be replaced by the current analog value. For this reason, we do not allow the characterization of a continuously changing, discretely existing flow and thus the table entry is left blank.

Discretely changing values can be characterized as digital values. Those that are continuously existing fit the previously mentioned shared variable analogy and are characterized as *persistent* flows. The flow value is not removed by a reference
to the flow, but rather the flow value persists over time until the source of the flow replaces the flow value with another. Those that are in existence only at discrete time periods, in particular from flow value creation to flow value use, are characterized as consumable flows. The flow value is removed by a reference to the flow.

Persistent and consumable flows describe the two most common types of value behaviors in software systems. However, software systems do interact with the real world, and real world values (for example temperature) are typically viewed as analog. SA specifications allow us to specify computer systems that might incorporate a mixture of hardware and software in the solution. This is often the case when developing distributed systems [47]. For example, an SA process that samples an analog temperature value and controls some mechanism based on the current temperature value could be implemented using a hardware device to convert the analog signal to a digital value and a software component that expresses the mechanism control. At the specification level, hardware implementation versus software implementation should be a non-issue. To specify the functionality, we assume the SA process is continuously monitoring the analog temperature value and reacts accordingly when that value falls into specific temperature ranges.

The three flow value behavior types: analog, persistent, and consumable, represent the potential behavior of flow values on a particular flow. Flow value behaviors are associated with flows and not DDEs. A particular label may be used to label more than one flow in a DFD. This does not mean that every flow with the same label must have the same flow value behavior. The DDE associated with a flow label just describes the domain, i.e. type, of potential flow values. The flow value behavior defines the behavior of elements of the domain on the particular flow. Thus flow
value behavior is an attribute of a flow and not a DDE. To model this attribute we add an additional component, \textit{FlowBehavior}, to the definition of FlowType as given in Type Definition 16.

\textbf{Type Definition 16}

\begin{verbatim}
FlowType = 4-tuple( FlowLabel : ModifiedLabel, 
                   Source : EndPoint, 
                   Destination : EndPoint, 
                   FlowBehavior : BehaviorType);
\end{verbatim}

BehaviorType is described in Type Definition 17 as an enumeration of analog, persistent, and consumable. We limit an analog flow's domain to either integer or real values, since analog behavior requires a continuous domain. This could be stated as an invariant property upon the completion of the definition of all modifications and enhancements.

\textbf{Type Definition 17}

\begin{verbatim}
BehaviorType = (analog,persistent,consumable);
\end{verbatim}

Having an interpretation of flow value occurrences facilitates the development of an operational semantics. We will interpret the appearance of flow values on flows similar to a marking of a Petri net [51]. In the next section we consider interpretations of flow abstraction as defined by the DDE composition operators. The semantics of the application of a composition operator for flow abstraction may depend on the flow value behavior defined for the flows involved in the abstraction. Flow abstraction will also contribute to the representation of a Petri net like marking of a DFD.
In this section we consider several possible interpretations of flow abstraction.
Flow abstraction provides a method by which a single flow may be used to represent
a collection of flows.

Flow abstraction occurs in two forms. The first form, which we discuss in Section
5.4.1, involves a graphical short-hand for representing multiple flows with either the
same source or destination. This short-hand notation allows flows to be depicted as
diverging or converging. We call this vergence flow abstraction. The second form,
which we discuss in Section 5.4.2, involves the balancing of in-flows and out-flows of
a DFD bubble with the off-page flows of the child process represented by the DFD
bubble. We call this decomposition flow abstraction.

We begin by considering interpretations for vergence flow abstractions. Vergence
flow abstractions are easier to interpret because the association of abstracted flows
and the abstract flow is explicitly given by a vergence point in a DFD. We then
use the interpretation of vergence flow abstractions to develop an interpretation of
decomposition flow abstractions. Decomposition flow abstractions are more difficult
to interpret because the association of abstracted flows and the abstract flow is not
explicitly given, but rather must be derived from the DDE of the labels of all the
flows involved in the decomposition, i.e., parent in-flows, parent out-flows, and child
off-page flows.

Flow abstraction is interpreted in terms of the DDE composition expressions
associated with the labels of the flows involved. Modified labels appear both as flow
labels and in DDE composition expressions. Modifiers were introduced in Section
5.2 as a method of flow distinction when two (or more) like-labeled flows repre-
sent different values, but have the same structure. Modifiers can also be used as a mnemonic convenience for distinguishing different sources or destinations for flows. Teamwork allows modifiers to be added and removed from flow labels as these flows undergo abstraction [29]. In effect, Teamwork treats an empty modifier as a wild card for equality comparison of flow labels. We also use this wild card interpretation for empty modifiers and thus redefine infix equality (=) for modified labels as given in Expression Definition 6. We assume this definition of equality carries over to all implicit use of equality such as in a subset relationship between sets of modified labels. Adopting this wild card interpretation of modified label equality allows us to converge and diverge modified flows into or out of unmodified flows.

Expression Definition 6

\[ \text{define } (X:\text{ModifiedLabel})'='(Y:\text{ModifiedLabel}) \text{ as boolean such that} \]
\[ '=' \equiv (\text{Identifier}(X) = \text{Identifier}(Y) \land \\
(\text{Modifier}(X) = \text{Modifier}(Y) \lor \text{Modifier}(X) = "" \lor \text{Modifier}(Y) = "")) \]

Any abstraction must have a starting point. Flow abstraction begins with flows that are not abstractions of other flows. We call these primitive flows. In terms of flow abstraction, primitive flows are flows that do not diverge and do not decompose in parallel with DFD bubble decomposition. These are the flows that would appear in the “football field sized” DFD, assuming vergence points are not allowed and decomposition flow abstraction between a DFD bubble and its corresponding P-Spec does not occur. These flows can be defined, as in Definition 1, as the flows labeled by the labels of P-Spec in-flows and out-flows. P-Spec in-flows and out-flows necessarily do not diverge or decompose further. Recall that each primitive flow has an associated
flow value behavior as defined in Section 5.3. Consideration of flow value behavior may influence the derived interpretations of flow abstraction.

**Definition 1**

A primitive flow is any flow \( f \) such that given an SA-Spec \( M \), \( \exists p \in Processes(M) \) and \( p \in \text{P-Spec} \) such that \( f \in (\text{InFlows}(p) \cup \text{OutFlows}(p)) \).

The DDE composition expression serves two roles. One role involves the description of potential values associated with a primitive flow or store label. Recall that this description of value can be interpreted as a description of type or of lexical value. We consider this issue in Chapter 6. The other role of DDE composition expressions involves the description of allowable flow abstractions. Our interpretations of flow abstraction requires that we view these two roles as fundamentally different. Those DDEs which directly or indirectly define the labels of primitive flows describe potential values. Those DDEs which define the labels of non-primitive flows, called composite flows, define the abstract composition of the composite flow. In this section we are concerned only with the second role of DDE composition expressions. In this section we are only interested in the DDE composition expressions associated with composite flows. To allow a more concise discussion of flow abstraction, in the remainder of this section we assume that all primitive flow labels are defined by elementary DDEs. Recall Type Definition 13 which defines a DDE definition as either elementary or composition. Under this assumption, only composite flow labels are defined by a DDE composition expression and DDE composition expressions are used only for describing allowable flow abstractions.
5.4.1 Vergence Flow Abstraction

Figure 5.10 shows the traditionally allowable labeling of convergence and divergence of flows [27, 59, 66]. Yourdon refers to a flow as describing the movement of packets of information from one part of the system to another [66]. Yourdon’s description of a flow is consistent with a consumable flow. Yourdon describes divergence as one of three possible situations: a copying of packets as in Figure 5.10(d), a splitting of a packet as in Figure 5.10(c), or a separating of packets based on value as also in Figure 5.10(c). Yourdon’s only description of convergence is, “Several elementary packets of data are joining together to form more complex, aggregate packets of data,” as in Figure 5.10(a). Ward says, “Flows of a particular type may converge or diverge to represent multiple sources, multiple destinations, or combination/separation of content” [59]. The particular type of flow depicted by Ward is the discrete-existence, i.e. consumable, flow. Ward would describe Figure 5.10(a) as two subsets of Z are supplied by two senders, (b) as all of Z can be supplied from either of two senders, (c) as two subsets of Z are sent to two receivers, and (d) as all of Z is sent to both of two receivers. Hatley and Pirbhai use a description similar to Ward, but also add two other labeling schemes shown in Figure 5.11. Hatley and Pirbhai would say the vergence abstractions in Figure 5.11 are allowed if X is an element of Z [27]. Recall that the unlabeled converging flow in Figure 5.11(a) and unlabeled diverging flow in (b) are implicitly labeled Z. In the case of Figure 5.11(a) it seems that the convergent flow Z now has two X values; one from the converging flow X and another from the converging flow implicitly labeled Z. Since X is an element of Z, an X value must already exist on the converging flow Z. Figure 5.11(b) might be representing a copying of only the X subordinate of Z, i.e., a copy of the X portion flows
on the diverging flow $X$ and the entire original $Z$ value flows on the diverging flow implicitly labeled $Z$. To understand the possible benefits and problems with vergence flow abstraction we examine several possible interpretations.

![Figure 5.10: Traditional Labelings of Vergence Flow Abstraction](image1)

![Figure 5.11: Other Labelings of Vergence Flow Abstraction](image2)

The Teamwork algorithm for balancing decomposition flow abstraction compares the set of child off-page in-flow labels with the set of parent bubble in-flow labels [29]. If all labels in both sets have not been matched then all the labels in the parent label set are broken down using the DDE composition expression.\(^1\) This process is iterated

\(^1\) If a label flagged as matched is broken down then the new elements created are also flagged as matched.
until all elements of the child label set and parent label set have been matched or
the parent label set cannot be broken down further. A similar process is followed
for balancing out-flows. In either case, only the parent flows are broken down during
the iteration. We call this type of balancing a strict balancing. The algorithm is
strict in the sense that in-flows can only decompose (diverge) and out-flows can only
compose (converge). A similar situation exists for vergence flow abstraction. We say a
convergent flow is a strict composition of the converging flows and the divergent flows
are a strict decomposition of a diverging flow if balancing requires only examining
the DD definition of the convergent or diverging flow. This strictness means that
a convergent flow labeled \( Z \), which is formed by converging flows labeled \( X \) and \( Y \),
must be defined in the DD in terms of \( X \) and \( Y \), not just in terms of the primitive flow
labels which define \( X \) and \( Y \). For example, assume \( A, B, C, \) and \( D \) label primitive
flows. Consider the following DD containing DDEs of \( R, U, V, W, X, Y, \) and \( Z \):
\[
\{(R, U + V), (U, A + C), (V, B + D), (W, A + B), (X, C + D), (Y, A + D), (Z, B + C)\}
\]
Suppose we wished to form a convergent flow labeled \( R \) from two flows labeled \( W \)
and \( X \). Using strict balancing a single convergence would not be allowed since \( R \) is
not defined in the DD in terms of \( W \) and \( X \). Using strict balancing, we would need
to diverge flows labeled \( W \) and \( X \) into flows labeled \( A, B, C, \) and \( D \). Flows \( A, B, C, \)
and \( D \) could then be converged directly into a flow \( R \) as shown in Figure 5.12. The
convergence of \( A \) and \( C \) into \( U, B \) and \( D \) into \( V, \) and \( U \) and \( V \) into \( R \) are all part
of the DD definition of \( R \). Similarly, diverging \( R \) into flows labeled \( Y \) and \( Z \) would
require 2 convergences to form \( Y \) and \( Z \).

When balancing convergences and divergences, most traditional methods are
strict, applying DDE composition expressions only to the convergent and diverging
flows. Similarly, when balancing decomposition flow abstraction in the DFD hierarchy, DDE composition expressions are applied only to in-flows and out-flows of the parent bubble. Commonly, most flow abstractions are strict; however, we do not wish to exclude consideration of non-strict abstractions as a shorthand for a cascade of convergences and divergences. In this section, we consider application of DDE composition expressions to both sides of a convergence or divergence. Thus the previous example might be depicted as a single non-strict convergence and a single non-strict divergence as shown in Figure 5.13. Simply put, non-strict convergences allow converging flows to implicitly diverge and divergent flows to implicitly converge.
However, we still have not yet defined a semantics of flow abstraction. Allowing non-strict vergence flow abstractions may limit the allowable forms and occurrences of vergence flow abstraction. In Section 5.4.1.1 we consider two interpretations of vergence flow abstraction that are independent of flow value behavior. In Section 5.4.1.2 we extend these interpretations using flow value behavior to characterize other possible interpretations.

5.4.1.1 Flow Bundling Abstraction: A simple non-flow value behavior approach to flow abstraction would be to consider convergence and divergence as a graphic bundling and unbundling of wires (flows). A primitive flow’s label corresponds to a wire’s colored insulation. Composite flow labels correspond to the colored insulating jacket that wraps a collection of wires together into a bundle. Each primitive flow (wire) retains its separate identity (colored insulation) inside the composite flow. To avoid ambiguity, no composite flow may contain two flows with the same label (colored insulation). Otherwise, when we take a composite flow apart we wouldn’t know which of the two flows to use. Each composite flow represents a union of all the primitive flows of which it is composed. Bundling requires only a single composition operator, we use +, and interpret + as the union of sets of primitive flow labels. A primitive flow label defines a singleton set composed of itself. Bundling assumes that + is the only operator used to construct DDE composition expressions for composite flows. However, we can take two approaches to describing this bundling union; either a non-disjoint union which we call simple bundling, or a disjoint union which we call layered bundling.
5.4.1.1.1 Simple Bundling: The convergence analogy for simple bundling is to strip each converging flow (bundle) down to a set of primitive flows (single wires). These wires are collected into one bundle and wrapped with a single colored insulating jacket, i.e., the label of the convergent flow. The convergent flow represents the union of the set of primitive flows with only a single colored insulating jacket wrapping the entire collection. Each primitive flow still retains its own colored insulation. Ambiguity results when two or more primitive flows contained in a composite flow have the same label.

Simple bundling is definitely a non-strict form of abstraction. Even though several subordinate DDEs may be used to define a single composite flow, the meaning of the flow is understood only in terms of the subordinate DDEs labeling the primitive flows contained in the composite flow. This holds true for both the decomposed flows (converging or divergent) and the composed flow (convergent or diverging). Simple bundling divergence strips a diverging flow into its primitive flows and disjointly partitions them for wrapping by the divergent flow labels.

Expression Definition 7

\[ \text{define } \text{PrimitiveLabels}(L:\text{ModifiedLabel}, DD: \text{set of DDE}) \text{ as set of ModifiedLabel such that} \]
\[ \exists d \in DD \land \text{Identifier}(L) = \text{Name}(d) \land \text{EntryDef}(d) \land \text{Composition} \]
\[ \land \text{PrimitiveLabels} = \{ L \} \]
\[ \lor \exists d \in DD \land \text{Identifier}(L) = \text{Name}(d) \land \text{EntryDef}(d) \land \text{Composition} \]
\[ \land \text{PrimitiveLabels} = \left( \bigcup_{l \in \text{ExpLabels(EntryDef)}} \text{PrimitiveLabels}(l, DD) \right) \]

We use Expression Definition 7 to describe the set of primitive flow labels defined by an arbitrary flow label. Recall our assumption that all primitive flow labels are de-
fined by elementary DDEs. The allowable occurrences of simple bundling abstraction in an SA-Spec $M$ are described as follows:

- Let $dd = DataDictionary(M)$.
- Let $Z$ be the label of the composed flow, i.e., the convergent flow or diverging flow.
- Let $X_1, X_2, \ldots , X_n$ be the labels of the decomposed flows, i.e., the converging flows or divergent flows.
- The $X$'s are composable if the primitive labels associated with any two distinct $X$'s are disjoint thus avoiding ambiguity. Formally:

$$\forall i \forall j[1 \leq i < j \leq n \Rightarrow \text{PrimitiveLabels}(X_i, dd) \cap \text{PrimitiveLabels}(X_j, dd) = \{ \}]$$

- $Z$ is a valid vergence flow abstraction of the $X_i$'s if:

$$\text{PrimitiveLabels}(Z, dd) = \left( \bigcup_{1 \leq i \leq n} \text{PrimitiveLabels}(X_i, dd) \right)$$

### 5.4.1.1.2 Layered Bundling:

The meaning of a composite flow using simple bundling was independent of the particular non-elementary subordinate DDEs used to define the composite flow's label. With layered bundling we use the non-elementary subordinate DDEs to augment avoidance of ambiguity in composite flows. Layered bundling is less restrictive as it allows two primitive flows with the same label to appear in a bundle as long as they are contained in two disjoint sub-bundles.
Layered bundling is analogous to a layering of colored insulating jackets. For example, suppose we are given primitive flow labels \( A, B, \) and \( C \). Suppose a composite flow labeled \( X \) is described as \( A + B \) and a composite flow labeled \( Y \) is described as \( B + C \). Then the two flows labeled \( X \) and \( Y \) could be converged into a single flow labeled \( Z \) assuming \( Z \) is described as \( X + Y \). The \( X \) version of \( B \) and the \( Y \) version of \( B \) are kept disjoint by their association with either \( X \) or \( Y \). Ambiguity exists in a composite flow only when the composite flow label's DDE contains two or more equal labels. To use the analogy, \( X \) is an insulating jacket wrapping \( A + B \) and \( Y \) is an insulating jacket (of a differing color) wrapping \( B + C \). \( Z \) is another insulating jacket that wraps \( X + Y \). However, we must also make sure that no ambiguity is exposed in a vergence point. For example, \( X \) and \( Y \) could not be converged into a flow labeled \( W \) where \( W \) was defined as \( A + B + C \) since \( W \) could only represent a single \( B \) flow. Also \( X \) and \( Y \) could not be converged into a flow labeled \( V \) where \( V \) was defined as \( P + Q \), \( P \) was defined as \( A + B \), and \( Q \) was defined as \( B + C \) because it is unclear whether \( X \)’s \( B \) is used for \( P \)’s \( B \) or \( Q \)’s \( B \).

Layered bundling convergence may strip converging flows as long as it does not expose an insulation color that occurs in any other sub-bundle of any sibling converging flow including itself. (All converging flows with the same vergence point are siblings.) Thus the converging flow serving as the source of any particular sub-bundle exposed in a convergence is uniquely determined. The flows exposed by stripping the converging flows are rebundled to form the convergent flow. Similarly, layered bundling divergence strips away colored insulating jackets from the diverging flow and rebundles the exposed flows to form the divergent flows. Allowing only the convergent and diverging flows to be stripped would represent strict decomposition. By
also allowing the converging and divergent flows to be stripped, we are defining a non-strict form of abstraction. This allows us to represent in a single non-strict convergence or divergence a cascade of strict divergences and convergences as described previously. Convergence and divergence points behave like gray boxes. We see the colors of the insulating jackets on wire bundles going in and out of a convergence or divergence point. The insides of the gray box are visible only through the DDE expressions associated with the flow labels. The insides of a gray box may actually contain several divergences and convergences as long as no ambiguities are exposed. (Ambiguity is the result of exposing a flow whose source or destination is not uniquely determinable.) The DDE expressions are examined to determine if the convergence or divergence can be constructed without exposing an ambiguity. Each in-flow diverges into sub-bundles according to its label's DDE composition expression. Each out-flow is a convergence of a disjoint set of sub-bundles defined by the in-flows and its label's DDE composition expression.

Allowable layered bundling convergences and divergences can be described using expression trees rooted by the label of a flow. Figure 5.14 provides an algorithm for constructing the expression tree, called a vergence tree, associated with a particular flow label $Z$, given a DD and an initially empty set of modified labels $Lset$. The formal parameter $Lset$ defines the set of all labels that would appear in the full expression tree if it were pruned to exclude the subtree rooted at $Z$. A vergence tree is essentially a pruned version of the full expression tree defined by the composition expressions in the DD. Recall that the only composition operator allowed by bundling is $+$ which represents union of sets of primitive labels. The vergence tree is formed from the full expression tree by pruning the descendants of any node which has an offspring
that appears more than once in the full expression tree, i.e., duplicate labeled nodes, their siblings, and descendants thereof are not allowed. To avoid ambiguity, the convergence or divergence must be describable in terms of these pruned expression trees. We use Expression Definition 8 to define the set of all modified labels used directly or indirectly in the definition of an arbitrary modified label. A vergence tree represents the maximum amount of insulation stripping that can be done without exposing a flow whose source or destination is ambiguously defined. Figure 5.15 provides an example of a full expression tree and its associated vergence tree.

Expression Definition 8

\[
\text{define } \text{AllLabels}(L: \text{ModifiedLabel}, DD: \text{set of DDE}) \text{ as set of ModifiedLabel such that}
\]

\[
\neg \exists d \in DD \land \text{Identifier}(L) = \text{EntryName}(d) \land \text{EntryDef}(d) \land \text{Composition}
\land \text{AllLabels} = \{L\}
\]

\[
\lor \exists d \in DD \land \text{Identifier}(L) = \text{EntryName}(d) \land \text{EntryDef}(d) \land \text{Composition}
\land \text{AllLabels} = \{L\} \cup \left( \bigcup_{I \in \text{ExpLabels} \text{EntryDef}(d)} \text{AllLabels}(I, DD) \right)
\]

Although a vergence tree describes the maximum amount of insulation stripping that can occur without exposing an ambiguity, we don’t necessarily need to strip all the way to the leaves of a vergence tree. We define a cover and a minimal cover in Definition 2 to describe a particular amount of abstraction, i.e., insulation stripping, for a composite flow.

Definition 2

Given a vergence tree \( \hat{Z} \), a set of modified labels \( W \) is a cover for \( \hat{Z} \) if \( W \) is a subset of the set of all \( \text{FlowLabels} \) contained in the nodes of \( \hat{Z} \), and for every leaf \( I \) of \( \hat{Z} \) there exists an \( m \in W \) such that \( m \) is the \( \text{FlowLabel} \) of a node on the path
function VergenceTree(Z: ModifiedLabel;
    DD:set of DDE;
    Lset:set of ModifiedLabel): VergNode;

where
    VergNode = 2-tuple(FlowLabel:ModifiedLabel,
        Children:set of VergNode);

begin
    if AllLabels(Z, DD) = {Z}
        then return (Z, {}):VergNode
    else
        find d ∈ DD such that EntryName(d) = Identifier(Z)
        ∧ EntryDef(d)\ Composition
        if ∃ x ∈ ExpLabels(EntryDef(d))
            ∧ x ∈ Lset ∪ (\bigcup_{y\in ExpLabels(EntryDef(d)) \land y\neq x} AllLabels(y, DD))
            then return (Z, {}):VergNode
        else
            let children = {}
            for each x ∈ ExpLabels(EntryDef(d)) do
                let newLset = Lset ∪ (\bigcup_{y\in ExpLabels(EntryDef(d)) \land y\neq x} AllLabels(y, DD))
                let children = children \cup {VergenceTree(x, DD, newLset)}
            endfor
            return (Z, children):VergNode
        endif
    endif
end VergenceTree

Figure 5.14: Vergence Tree Construction Algorithm
Full Expression Tree

```
    Z
   /\  
  X   Y
 /   /|
A   S  U
|    |
E   F  D
```

Vergence Tree

```
    Z
   /\  
  X   Y
      /|
     W  V
       |
        E F B D
```

Figure 5.15: Example Vergence Tree
from l to the root of \( \hat{Z} \). \( W \) is a \emph{minimal cover} if \( W \) is a cover, and for every leaf \( l \) of \( \hat{Z} \) there exists a unique \( m \in W \) such that \( m \) is the FlowLabel of a node on the path from \( l \) to the root of \( \hat{Z} \).

The smallest minimal cover is the set containing the label of the root of the vergence tree. The largest minimal cover is the set containing the labels of the leaves of the vergence tree.

We describe allowable layered bundling abstraction in an SA-Spec \( M \) as follows:

- Let \( dd = \text{DataDictionary}(M) \).
- Let \( Z \) be the label of the composed flow.
- Let \( X_1, X_2, \ldots, X_n \) be the decomposed flow labels.
- Construct the vergence tree for \( Z \), i.e., let \( \hat{Z} = \text{VergenceTree}(Z, dd, \{ \}) \).
- Construct the vergence tree for a fictitious flow label \( Y \) defined as \( X_1 + X_2 + \cdots + X_n \), i.e., let \( \hat{Y} = \text{VergenceTree}(Y, dd \cup \{(Y, X_1 + X_2 + \cdots + X_n)\}, \{\}) \).
- The \( X \)'s are composable if \( \forall i, \forall j, (1 \leq i < j \leq n \Rightarrow X_i \neq X_j) \), i.e., the \( X \)'s themselves do not cause an ambiguity. Note this means that every \( X_i \) labels a node in the vergence tree \( \hat{Y} \).
- \( Z \) is a valid vergence flow abstraction of the \( X \)'s if there exists a set of modified labels \( W \neq \{ Y \} \) such that \( W \) is a minimal cover of both \( \hat{Y} \) and \( \hat{Z} \). \( W \) represents the amount of insulation stripped from the \( X_i \)'s and from \( Z \) in order to form the abstraction. Although \( W \) is not necessarily unique, if a minimal cover does exist for \( \hat{Y} \) and \( \hat{Z} \) then there does exist a unique \emph{smallest minimal cover} for \( \hat{Y} \).
and \( \hat{Z} \) and we choose \( W \) as this smallest minimal cover. (Smallness is defined in terms of set cardinality.) The uniqueness of this smallest minimal cover is provided by Theorem 1. If \( W = \{X_1, X_2, \ldots, X_n\} \) then this is an example of a strict vergence.

**Theorem 1**

Let \( \hat{Y} \) and \( \hat{Z} \) be two vergence trees as described by layered bundling abstraction. If there exists a minimal cover for \( \hat{Y} \) and \( \hat{Z} \) (not equal \( \{ Y \} \)) then there exists a unique smallest minimal cover for \( \hat{Y} \) and \( \hat{Z} \) (not equal \( \{ Y \} \)).

**proof:** (by contradiction)

Since \( Y \) is fictitious, \( Y \) cannot appear as a FlowLabel in \( \hat{Z} \), and thus \( \{ Y \} \) cannot be a cover for \( \hat{Z} \). Let \( W_1 \) and \( W_2 \) be two distinct smallest minimal covers of \( \hat{Y} \) and \( \hat{Z} \). Since \( W_1 \neq W_2 \), \( \exists x \in W_1 \) such that \( x \notin W_2 \) and without loss of generality \( \exists y_1 \ldots y_n \in W_2 (n \geq 2) \), such that \( x \) lies on every path from the root to a leaf on which a \( y_i \) lies in both \( \hat{Y} \) and \( \hat{Z} \) and \( x \) does not lie on a root to leaf path shared by any other elements of \( W_2 \). These labels \( x, y_1, \ldots, y_n \) exist because \( \hat{Y} \) and \( \hat{Z} \) are constructed from the same DD. So \( \{ x \} \) and \( \{ y_1, \ldots, y_n \} \) are covering the same subtrees in \( \hat{Y} \) and \( \hat{Z} \). Then \( W_3 = (W_2 - \{ y_1, \ldots, y_n \}) \cup \{ x \} \) is also a minimal cover for both \( \hat{Y} \) and \( \hat{Z} \) and \( W_3 \) is smaller than both \( W_1 \) and \( W_2 \). Thus \( W_1 \) and \( W_2 \) cannot both be smallest minimal covers which is a contradiction. \( \blacksquare \)

Both the simple and layered bundling interpretations of vergence flow abstraction do not require bundled flows to have the same flow value behavior. Bundling is just a graphical short-hand notation for drawing multiple flows. However, bundling does limit DDE composition expressions to use only the + operator. Since flows are static
representations of data, i.e., flows do not disappear or reappear, it doesn't make sense to have a flow bundling abstraction that sometimes represents a flow labeled $X$ and other times represents a flow labeled $Y$. Thus a flow labeled $Z$ defined as $[X \mid Y]$ doesn't make sense from a flow bundling perspective. However, we might interpret $Z$ as a flow that sometimes has a value from the domain defined by $X$ and other times has a value from the domain defined by $Y$. We call this type of interpretation a flow value interpretation of vergence flow abstraction. In the following section we build on the bundling interpretation of vergence flow abstraction to describe this flow value interpretation of vergence flow abstraction.

5.4.1.2 Flow Value Abstraction: The flow value interpretation of vergence flow abstraction provides a semantically richer notion of abstraction of flow values, rather than a graphical abstraction of flows. This approach provides similar semantics to those suggested by Ward for consumable valued flows [59]. However, we must be careful when we apply these semantics to flows of differing flow value behavior. In this section, we only consider convergence and divergence of flows with the same flow value behavior. For example, we would not consider a convergence of a persistent flow and a consumable flow.

5.4.1.2.1 Analog Flow Abstraction: Consider two primitive analog flows labeled $X$ and $Y$. If $X$ and $Y$ are represented as converging into a flow labeled $Z$, how do we interpret $Z$ in terms of the analog values of $X$ and $Y$? If the flows were consumable flows, Ward would describe this as, "Two subsets of $Z$ are supplied by two senders." Does the value of $Z$ represent a function of the values provided by $X$ and $Y$? If so, what function?
Consider the convergence of two unlabeled primitive analog flows into a single primitive analog flow labeled $Z$. The two unlabeled flows are implicitly also labeled $Z$. What is the interpretation of the convergence of two analog values from the same domain? If the flows were consumable flows, Ward would describe this as, "$Z$ is supplied by either of two sources." Is the convergent flow a function or selection of the two converging flows. If a function, what function? If a selection, based on what?

In either case each analog flow represents (or carries) a single analog value which is constantly changing over time. Combining two analog values into a single analog value without some description of the function that combines them is nonsense. Such a function would most likely be non-trivial and should thus be represented by a bubble. In the first case, $Z$ must in some way represent both analog values, $X$ and $Y$, in some non-combined form. Since analog values are associated one-to-one with analog flows, a composite analog flow represents multiple flows much like the simple bundling approaches considered previously. In the second case, we have no generalized method for selecting one of the two converging $Z$s to represent the value of the convergent $Z$. They are both changing constantly, so we can't really say to use the value that changed most recently. This type of convergence just doesn't make sense, unless we interpret a composite analog flow as individually representing each analog value, i.e., primitive analog flow. Thus analog convergence can be interpreted exactly like simple bundling.

Analog divergence can also be interpreted similarly to simple bundling. The only difference is that from a flow value perspective we can also consider the duplication or copying of a single analog value onto two or more separate analog flows labeled by the

---

2Such a function would also most likely be implemented in hardware.
same label. In essence we are viewing an analog value as having infinite signal strength and this allows us to slice off copies of the analog value. Applying this divergence duplication to the allowable forms of simple bundling relaxes the restriction requiring the $X_i$'s to be disjoint when they are divergent flows.

A layered bundling interpretation of convergence and divergence of analog flows can be defined so as to maintain the same relationship between simple and layered bundling. Since in-flows to a convergence point may undergo divergence before converging to form the out-flow, the cover associated with a convergence point need only be a minimal cover for the out-flow vergence tree. This guarantees unique sources for each out-flow represented as an element in the cover. Where the cover is not minimal on the in-flow vergence tree, divergence duplication is taking place. We generalize a layered bundling version of analog convergence in an SA-Spec $M$ as follows:

- Let $dd = DataDictionary(M)$.

- Let $X_1, X_2, \ldots, X_n \ (n \geq 2)$ be the labels of the converging analog flows and let $Z$ be the label of the convergent analog flow.

- Let $\hat{Y} = VvergenceTree(Y, dd \cup \{(Y, X_1 + X_2 + \cdots + X_n)\}, \{\})$, for a fictitious flow label $Y$ defined as $X_1 + X_2 + \cdots + X_n$ and let $\hat{Z} = VvergenceTree(Z, dd, \{\})$.

- The $X$'s are composable if $\forall i, \forall j, (1 \leq i < j \leq n \Rightarrow X_i \neq X_j)$, i.e., the $X$'s themselves do not cause an ambiguity. Note this means that every $X_i$ labels a node in the vergence tree $\hat{Y}$.

- The convergence is valid if there exists a set of modified labels $W$ such that $W$ is a minimal cover of $\hat{Z}$ and $W$ is a cover for $\hat{Y}$. The actual balancing
of flows is defined by the smallest set of modified labels $W$ that satisfies this property. An element which appears in $W$ and is closer to a leaf in $\hat{Y}$ than the first occurrence of an element of $W$ on the path from the root of $\hat{Y}$ to that leaf, represents duplication of part of a sub-bundled analog flow. No direct multiple copies of a particular analog flow are possible with convergence, only copies of sub-bundles, since the convergent flow can only represent one flow value for any particular flow label.

We allow multiple duplication of values on different diverging flows in a divergence. To do this, we consider a separate vergence tree for each separate diverging flow, rather than a fictitious flow defined as the union of the diverging flows. Thus each element of the cover on the in-flow vergence tree may actually be an element in several different out-flow vergence trees, representing multiple destinations for the analog value. We generalize a layered bundling version of analog divergence in an SA-Spec $M$ as follows:

- Let $dd = DataDictionary(M)$.
- Let $X_1, X_2, \ldots, X_n$ be the divergent analog flow labels and $Z$ be the diverging analog flow label.
- Let $\hat{X}_i = VergenceTree(X_i, dd, \{\})$ for all $i, 1 \leq i \leq n$ and let $\hat{Z} = VergenceTree(Z, dd, \{\})$.
- The divergence is valid if there exists sets of modified labels $W_1, W_2, \ldots, W_n$, not necessarily distinct or disjoint, such that $W_i$ is a minimal cover of $\hat{X}_i$ and
\[ W = \left( \bigcup_{1 \leq i \leq n} W_i \right) \]
is a cover for \( \hat{Z} \). The actual balancing of flows is defined by the smallest such sets \( W_i \). As with layered bundling convergence, an element which appears in \( W \) and is closer to a leaf in \( \hat{Z} \) than the first occurrence of an element of \( W \) on the path from the root of \( \hat{Z} \) to that leaf, represents duplication of part of a sub-bundled analog flow. An element which appears in more than one \( W_i \) represents duplication of an entire analog flow value.

Analog flow abstraction adds to the bundling interpretation only the duplication of value when a flow diverges. The effect on the “football field sized” DFD is that each primitive analog flow has a unique source, but may have multiple destinations. Each destination has access to the analog value represented by the flow, but change to the analog value is controlled by the single unique source.

Analog flow abstraction is still just a graphic abstraction for multiple flows, hence we allow only the + composition expression operator for defining composite analog flow labels. Assuming the alternative composition expression operator defined a choice of value domains, then allowing the alternative composition expression operator for analog flow abstraction would require a generalization of a function that made the choice of which analog value to currently use as the value for the composed flow. We know of no such generalization.

5.4.1.2.2 Persistent Flow Abstraction: Persistent flow abstraction is similar to analog flow abstraction. Two like-labeled, primitive, converging, persistent flows represent two continuously existing values from the same domain. As with
two primitive, converging, analog flows, we cannot provide a generalized function for combining the two converging flow values into a single value on the convergent flow. Likewise, a primitive persistent flow could diverge into multiple copies of its value. Thus we could adopt semantics similar to analog flow abstraction for persistent flow abstraction. However, a persistent flow is not continuously changing, but rather changes at discrete points in time. If we adopt an interpretation similar to multiple writers of a shared variable, then we can relax the restriction requiring unique sources for convergence. Thus the generalization for combining two values is to choose the value that was changed most recently. This is not a function but rather a process as it exhibits time dependent behavior. Note that if a process "writes" the same value on to a primitive persistent flow more than once in a row, we consider each "write" as a change in the value of the flow.

For simple bundling persistent convergence, this means relaxing the restriction requiring

\[ \forall i, \forall j [1 \leq i < j \leq n \Rightarrow \text{PrimitiveLabels}(X_i, dd) \cap \text{PrimitiveLabels}(X_j, dd) = \{ \}]. \]

In the case where

\[ \text{PrimitiveLabels}(X_i, dd) \cap \text{PrimitiveLabels}(X_j, dd) \neq \{ \} \]

then the primitive persistent flows common to both \( X_i \) and \( X_j \) represent multiple writers to the representation of these same primitive persistent flows in \( Z \).

We can model this multiple writer convergence with a bubble sitting at the point of convergence. Each primitive, converging, persistent flow common to two or more \( X_i \)'s is modeled by a primitive, converging, consumable flow of the same type, i.e., label. We model a "write" of a value to one of these primitive, converging, persistent
flows as a message sent on the corresponding primitive, converging, consumable flow. When the convergence bubble receives a message, it “writes” the value of the message onto the primitive, convergent, persistent flow. If two values from different primitive, converging, consumable flows arrive at exactly the same time then a nondeterministic choice is made as to which one arrived first; however, each arrival triggers the convergence bubble to write once. This convergence process bubble serves as our generalization for combining two (or more) persistent flow values from the same domain. This generalization was not possible with analog flows because each analog flow value was continuously changing, rather than changing at discrete time points.

Simple bundling persistent divergence is exactly analogous to simple bundling analog divergence. Two (or more) like-labeled, primitive, divergent, persistent flows represent copies of the primitive, diverging, persistent flow's value.

With layered bundling persistent convergence, the goal is to strip away as little insulation as possible and to not expose ambiguities in a single converging flow. This forces multiple writes to come from different converging flows and not from the particular composition of a single converging flow. The formalization of disjoint (i.e., layered bundling) persistent flow convergence is exactly the same as disjoint analog flow divergence with the direction of flow reversed.

For layered bundling persistent flow divergence, the analogy to analog flows is exact. Note that this preserves our intuitive analogy between persistent flows and shared variables in a concurrent system. Convergence represents multiple writers and divergence represents multiple readers.

What about combining two or more persistent flow values from different domains using the alternative composition expression operator? Consider a composed
persistent flow $Z$, where $Z$ is defined in the DD as $\{ X \mid Y \}$. (Assume $X$ and $Y$ label the primitive persistent flows used to compose $Z$.) Using the previous multiple writers analog, $Z$ represents a shared variable whose domain is a disjoint union of the domains of $X$ and $Y$. Every time a writer writes to the $X$ flow, the new $X$ value replaces the old value on the $Z$ flow; likewise for $Y$. Eventually, however, $Z$ must diverge into $X$ and $Y$. When $Z$ carries a $Y$ domain value, what is the value on the divergent $X$ flow? Since the divergent $X$ flow is persistent, it must have a value. If the value of the divergent $X$ flow is simply the last $X$ value written onto $Z$ then there is no difference between $\{ X \mid Y \}$ and $X + Y$ as the definition of $Z$. Thus the only composition operator allowed for persistent flow abstraction is $\_+$.

The effect of our interpretation of persistent flow abstraction on the "football field sized" DFD is to allow multiple sources and multiple destinations for any persistent flow. Multiple sources represent multiple writers, and multiple destinations represent multiple readers. In terms of allowing each writer to also access (i.e., read) the value of the shared variable there are three possibilities:

1. Assume each writer has implicit access to the current value of the shared variable to allow updates based on previous value. The flow out of a convergence point represents the value of the shared variable. If that flow converges with other flows then the next converged flow represents a second shared variable which may not always have the same value of the first shared variable. A similar situation arises when a diverged copy of a flow converges with other flows. This type of situation becomes difficult to interpret.

2. Assume each writer only has implicit access to the last value written by itself. Thus a writer can only update its own last value. To have access to the shared
variable would require a diverging flow directed back to the writer. For example, in Figure 5.16 only bubble Writer1 has access to the value of the shared variable represented by flow $X$. However, both Writer1 and Writer2 have implicit access to their own last value written.

![Diagram of flows](image)

Figure 5.16: No Implicit Access to the Convergent $X$ Flow

3. Assume each writer does not have access to any old values unless explicitly represented by diverging flows. A flow that diverges from a converging flow would only have access to the most recent value written by the source of the converging flow. A flow that diverges from a convergent flow would have access to the most recent value written by any of the sources of the convergent flow. For example, in Figure 5.17 bubble Writer1 has access to the last value written by any of the writers, while bubble Writer2 has access to only the last value written by itself, and bubble Writer3 does not have access to any previous values written.

We suggest the use of the third approach since it represents all flow value accesses
Figure 5.17: No Implicit Access to Any X Flows
explicitly. This is one example of the use of self looping flows previously mentioned in Chapter 4.

5.4.1.2.3 Consumable Flow Abstraction: Both analog and persistent flow abstractions are somewhat similar to the graphic bundling abstraction described for flows in general. This is because the correspondence between flows and flow values for both analog and persistent flows is one-to-one. However, this is not the case for consumable flows. A consumable flow represents a FIFO ordered queue of flow values. Thus the correspondence between a consumable flow and flow values is a one-to-many relationship.

The semantics of consumable flow abstraction includes a stronger dependence on the concept of time. The number of flow values associated with a consumable flow changes over time. During an operational interpretation, i.e., execution, of an SA specification, consumable flow values are produced by the flow's source and consumed by the flow's destination. A consumable flow value can be accessed only if it exists, i.e., has been produced but not yet consumed. When a consumable flow value is accessed it is consumed, i.e., removed from the flow, and cannot be accessed again. Analog and persistent flow values are always accessible and can be accessed repeatedly over time. Interpreting consumable flow vergence abstraction must take into consideration this one-to-many time-dependent relationship between consumable flows and consumable flow values.

Recall Ward's interpretation of the convergences and divergences of discrete-existence flows [59]. Ward describes the convergence of two flows, both labeled $Z$, into a single flow labeled $Z$ as, "All of $Z$ can be supplied from either of two senders."
For consumable flow values, this can be taken to describe a (fair) merging of flow values from the two sources. In Figure 5.18 we depict consumable flow values as small labeled circles called tokens. The convergence on the left of the double arrow represents the consumable flow values before the merge. The convergence on the right of the double arrow represents the consumable flow values after the merge. Extending this to our message channel analogy, messages arriving at the point of convergence are passed on to the convergent flow in a first-come, first-serve, order. Simultaneous arrivals are arbitrarily ordered. The actual merging of flow values happens as the flow values are produced. In Chapter 9 we include the abstraction of flow values in the execution of the producing bubble. When the execution of a bubble produces a consumable flow value, the flow value is “pushed” through all convergences and divergences possible as part of the execution of the producing bubble. Simultaneous arrivals at a convergence are possible only if the same bubble produced both arrivals in the same execution, or if we allow parallel execution of bubbles.

Ward describes the divergence of a discrete existence flow labeled Z into two flows both labeled Z as, “All of Z is sent to both of two receivers.” Ward’s description applies to all three flow behavior types. This type of consumable flow divergence is a duplicate copy divergence similar to the copy divergence defined for both analog
and persistent flows. When a consumable flow value arrives at the point of divergence, a copy of the flow value is passed on to each divergent flow. (See Figure 5.19.) The original FIFO ordering is maintained on each divergent flow; however, the copies on each divergent flow may be consumed at different rates by their respective destinations.

Ward describes the convergence of two discrete-existence flows labeled $X$ and $Y$ into a single flow labeled $Z$ as, "Two subsets of $Z$ are supplied by two senders." A flow value on $Z$ might be interpreted as a combination of two flow values, one from $X$ and one from $Y$. We would then require $Z$ to be defined in the DD as $X+Y$. In this case, we are describing a $Z$ value as both an $X$ value and a $Y$ value. Because we are abstracting flow values and not the actual flow, we need to also check that no ambiguity occurs in the balancing of $X+Y$ with the DDE definition of $Z$. This will depend on the allowance of non-strict abstraction. The convergence point represents a meeting place for $X$ values and $Y$ values. Whenever both an $X$ value and a $Y$ value are available, they are combined into a $Z$ value and passed on to the convergent flow. If a $Y$ value is not available when an $X$ value arrives, then the $X$ value queues up at the convergence point and waits for a $Y$ value. (See Figure 5.20.)

However, another interpretation can be described for the convergence of two

---

**Figure 5.19: The Copying Divergence**

- **Diagram:**
  - Source: $3, 2, 1$
  - Divergence: $c$
  - Destination: $3, 2, 1$
consumable flows labeled $X$ and $Y$ into a single flow $Z$. We could interpret this to mean that a $Z$ value is either an $X$ value or a $Y$ value. This form of convergence can be defined in the DD using the alternative composition expression operator. Thus in the DD, $Z$ could be defined as $[X \mid Y]$ and the convergence point represents an alternative merging of the two different kinds of $Z$s. (See Figure 5.21.) The message channel analogy can be extended to view $Z$ as a multiplexed message channel of $X$ and $Y$ messages.

Ward describes the divergence of a discrete-existence flow labeled $Z$ into two flows labeled $X$ and $Y$, respectively, as, “Two subsets of $Z$ are sent to two receivers.” For both analog and persistent flows, this represents an unbundling of flows with copies of the flows that appear in more than one diverging flow. This form of divergence extends naturally to consumable flows if the DD definition of $Z$ is $X + Y$. 
Each flow value on $Z$ is split apart into its subordinates $X$ and $Y$ with copies made of the subordinates $X$ and $Y$ have in common. (See Figure 5.22.)

![Figure 5.22: The Splitting Divergence](image)

However, suppose the definition of $Z$ is $[X \mid Y]$. Then each $Z$ flow value is either an $X$ value or a $Y$ value. When a $Z$ flow value arrives at the divergence point, it flows exclusively onto the divergent flow labeled by the kind of value it is. (See Figure 5.23.) This form of divergence represents a selection of destination based on type.

![Figure 5.23: The Selecting Divergence](image)

Recall Hatley and Pirbhai's additional possible labelings of vergences given in Figure 5.11 [27]. The convergence of an $X$ flow and a $Z$ flow into a $Z$ flow can be interpreted in two ways. If $Z$ is defined as $[X \mid \ldots]$ then the convergence can represent a composition of a select, a merge, and an alternate vergence. If $Z$ is defined as $X+\ldots$ then the convergence can represent a composition of a copy, a split, a combine, and a merge vergence. In the first case the timing is obvious. (See Figure
5.24.) Flow values appearing on either converging flow merge onto the convergent flow. In the second case there are several choices. As discussed previously, such a convergence might not be allowed. A converging Z flow value already contains an X flow value and we have no generalized function to combine two X values into one. However, we could interpret this to mean that a copy is made of all but the X portion of a converging Z flow value and this copy is combined with the converging X flow value. The original Z flow value flows onto the convergent flow. In this case we also need to decide if only X flow values will queue up waiting for copies of part of a converging Z flow value or if copies of part of a converging Z flow will also queue up to wait for X flow values. The latter determination seems more customary and can be modeled by the existing vergences as in Figure 5.25.

![Diagram](image)

**Figure 5.24:** Timing of a Composite Select/Merge/Alternate Vergence

![Diagram](image)

**Figure 5.25:** Timing of a Composite Copy/Split/Combine/Merge Vergence
The divergence of a $Z$ flow into an $X$ flow and a $Z$ flow can also be interpreted in two ways. If $Z$ is defined as $\{X | \ldots\}$, then the divergence represents a composition of a select, copy, and alternate vergence. If $Z$ is defined as $X+\ldots$, then the divergence might represent a composition of a copy and split divergence. Again in the first case the timing is clear. (See Figure 5.26.) When a $Z$ flow value is an $X$ value it copies, otherwise it selects only the diverging $Z$ flow. In the second case each $Z$ flow value copies. One of the copies splits into its $X$ portion and the rest of that copy is discarded. (See Figure 5.27.)

![Figure 5.26: Timing of a Composite Select/Copy/Alternate Vergence](image)

Except for the composite vergences, the above description of convergences and divergences represent *simple* vergences—only one type of abstraction is taking place. For convergences we have three types of abstractions: merge, combine, and alternate. For divergences we also have three types of abstractions: copy, split, and select.
In a combine, an alternate, a split or a select vergence the DDE definition used only one composition expression operator. The merge and copy vergences are DDE independent, and the graphical depiction of these vergences do not imply a particular order among the converging or divergent flows. Thus we assume the combine/split and alternate/select composition expression operators are commutative. The time dependent behavior of flow values in consumable flow vergence abstractions can also be shown to be associative. For example, the composition expression $W+X+Y$ is equivalent to $X+Y+W$.\(^3\) Similarly, $[W|[X|Y]]$ is equivalent to $[[W|X]|Y]$. Because they are associative, we tend to view the $+$ and $[\ldots]$ operators as n-ary operators. However, this associativity will not extend across DDE definition substitution in layered bundling. For example, suppose $Z$ is defined as $A+X$ and $X$ is defined as $Z$. $Z$ cannot be allowed to be equivalent to $A+A+B$ because of the ambiguity problem.

The primary difference between simple bundling and layered bundling is the definition of ambiguity in a composite flow. The balancing of flow abstractions assures that ambiguity is not introduced into a composite flow. With consumable flow abstractions, ambiguity is defined as a composite flow value that contains two composing flow values from the same domain. For example, we would not allow a combine convergence of two flows both labeled $X$ into a single flow labeled $Z$ where $Z$ was defined in the DD as $X+X$. (See Figure 5.28.) A flow value on $Z$ would be composed of two $X$ values. When $Z$ split diverged, which $X$ value would go on which diverging $A'$ flow? A similar situation exists for alternate converge and select diverge.

\(^3\)Without parenthesis, associativity is shown as a reordering of operands as read from left to right.
The difference between simple bundling ambiguity and layered bundling ambiguity is whether we allow primitive flow values to be encapsulated by a nesting of DDEs. For example, consider again the combine converge given in Figure 5.20. Assume the following DD:

\{ (Z, X + Y), (X, A + B), (Y, B + C) \}

There are two \( B \) values in each \( Z \) flow value. With simple bundling the flow values on \( Z \) would be ambiguous with respect to \( B \). With layered bundling the flow values on \( Z \) are not ambiguous because each \( B \) value is encapsulated by either \( X \) or \( Y \). Thus with consumable flows avoiding ambiguity is a restriction on DDEs.

The previous consumable vergence abstractions involved at most a single composition expression operator (copy and merge do not involve an operator). However, the syntax of DDEs do allow the nesting of operators within a single DDE. First we examine the effect of nesting on domain composition for the determination of ambiguous DDEs. Then we provide the flow value timing of abstractions involving nesting of composition expression operators.

To describe the possible domain composition of a composite flow value we need only consider a disjunctive normal form (DNF) of DDEs composition expressions. We assume \(+\) and \([\ldots]\) are the only operators used to define consumable flow labels.
For domain composition consideration, + distributes over \([\cdot \mid \ldots]\), e.g., \(W + [X \mid Y] \equiv [W + X \mid W + Y]\) with respect to flow value domain composition. Under layered bundling a consumable flow value's DDE is ambiguous if a single label appears more than once in any conjunct in the DNF. For example, \([A \mid B] + [A \mid C]\) is ambiguous because the DNF, \([A + A \mid A + C \mid B + A \mid B + C]\), contains a conjunct with two As. Under simple bundling a consumable flow value's DDE is ambiguous if after repeated substitution of DDE definitions until all labels are primitive, the DNF contains a conjunct in which a single label appears more than once. For example, even though \(X + Y\) is not layered bundling ambiguous, if \(X\) is defined as \([A \mid B]\) and \(Y\) is defined as \([A \mid C]\) then the resulting substitutions for \(X\) and \(Y\) result in the same ambiguous DNF given previously.

Nesting of composition expression operators allows for the representation of non-simple vergences, i.e., a vergence in which more than one type of abstraction is taking place. This results from the nesting of + and \([\cdot \mid \ldots]\) operators in a single DDE composition expression. For example, consider Figure 5.29. Suppose three flows \(W\), \(X\), and \(Y\) converge into a composite flow \(Z\). Further suppose \(Z\) is defined in the DD as \(W + [X \mid Y]\). This non-simple convergence is equivalent to the cascade of simple convergences shown in Figure 5.30. \(X\) and \(Y\) alternate converge and then this flow combine converges with \(W\).

However, suppose \(Z\) is defined in the DD as \([W + X \mid W + Y]\). Does combine distribute over alternate with respect to the timing of flow values? One possible interpretation of Figure 5.29 using this DDE is given in Figure 5.31. Since the \(W\) appears twice in \(Z\), \(W\) copy diverges, then each copy combine converges with either \(X\) or \(Y\), and then these combined flows alternate converge into \(Z\). Using
Figure 5.29: A Non-simple Convergence

Figure 5.30: Equivalent Simple Convergences
this interpretation, combine convergences clearly do not distribute over alternate convergences with respect to timing. Consider the flow value timings given in Figures 5.32 and 5.33. In Figure 5.32 the \( y2 \) value (which we assume arrived last) queues up at the combine convergence and waits for a \( W \) value. The flow values on \( Z \) are \((w1,x1), (w2,y1), \) and \((w3,x2)\). In Figure 5.33 the two copies of \( w3 \) queue up at the two combine convergences and wait for \( X \) and \( Y \) values. The flow values on \( Z \) are \((w1,x1), (w1,y1), (w2,x2), \) and \((w2,y2)\). Note that the flow values on \( Z \) have the same domain structure—they are either a \( W \) and an \( X \) value, or a \( W \) and a \( Y \) value. Recall that combine does distribute with alternate when considering domain composition.

![Figure 5.31: Equivalent Simple Vergences](image)

The difference in timing is caused by our assumption that because \( W \) appeared twice in the second definition of \( Z \), \( W \) must copy diverge. An alternative would be to interpret multiple appearances of flow labels in a DDE in a demand driven manner. Rather than interpreting the double appearance of \( W \) in the DD definition of \( Z \) as a copy divergence, we could imagine that \( W \) flow values wait at a demand divergence point until a \( W \) value is required on one of the diverging flows. When a \( W \) value
Figure 5.32: Timing of Figure 5.30

Figure 5.33: Timing of Figure 5.31
is required, for example an \( X \) value arrives, then if available a \( W \) value flows onto only one of the diverging flows. A demand is not fulfilled unless the demand will produce an out-flow value for the entire vergence point. Note that it is possible that more than one in-flow value is demanded to fulfill a demand and that there may be a non-deterministic choice of which demand to fulfill.

Using this interpretation, combine convergences do distribute over alternate convergences. However, demand divergences are not one of the simple divergences, which are based on traditional descriptions of flow divergence, discussed previously. Thus if we allow non-simple vergences, should demand divergences be used to interpret multiple label appearance in DDEs (giving us distributivity of combine over alternate). Furthermore, should demand divergences be added to our list of simple divergence abstractions (requiring a graphic notation to distinguish them from copy)? Traditionally the + operator is described as and and the \([. . .]\) operator as exclusive-or. In boolean logic, and does distribute over exclusive-or. Note that alternate convergence does not distribute over combine convergence. Consider the timing of \([W \mid X]+[W \mid Y]\) and \([W \mid X+Y]\).\(^4\) This also holds true for boolean logic; exclusive-or does not distribute over and.

Consider the layered bundling example in Figure 5.34. Assume the following DD:

\[
\{(Z, [X + Y \mid X + W \mid Y + W]), (X, A + B), (Y, A + C), (W, B + C)\}
\]

The composition of any particular \( Z \) flow value will be two \( A \), \( B \), or \( C \) flow values and one flow value from each of the other two, e.g., \( AABC \), \( ABBC \), or \( ABCC \).\(^5\) The

\(^4\)The former expression is actually an ambiguous domain composition.

\(^5\)We assume a layered bundling interpretation to avoid ambiguity.
simple vergence equivalent is given in Figure 5.35. However, three different timing interpretations are possible:

1. all "?" labeled divergences are demand driven;

2. the multiple appearance of labels in $Z$ are demand driven divergences (i.e., the "?" labeled divergences on the left), but the $A$, $B$, and $C$ are copy divergences (i.e., the "?" labeled divergences on the right); or

3. three, all "?" labeled divergences are copy.

In the first interpretation, layered bundling provides only a method for avoiding ambiguity in domain composition. Timing distributivity extends across nested layers for DDEs. In the second interpretation, layered bundling abstracts the copy divergence and timing distributivity applies only within the context of a single DDE. In the third interpretation, timing distributivity does not hold. Table 5.2 provides a comparison of out-flow values resulting from each interpretation applied to the same in-flow values.

There is no basis in the current literature for a decision of which interpretation to use. We suggest the second because it allows for a larger set of possible timing representations. If you want copy, bury the copies in DDE substitution. The copy
Table 5.2: Layered Bundling Timing Example

In-flows in Order of Appearance

\[ a_1, b_2, c_3, c_4, a_5, b_6, b_7, c_8, a_9 \]

<table>
<thead>
<tr>
<th>The Three Timing Interpretations of Out-flows</th>
</tr>
</thead>
<tbody>
<tr>
<td>demand,demand</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>((a_1, c_3), (b_2, c_4))</td>
</tr>
<tr>
<td>((a_5, b_7), (b_6, c_8))</td>
</tr>
<tr>
<td>((a_5, b_6), (b_6, c_4))</td>
</tr>
<tr>
<td>((a_9, b_7), (b_7, c_8))</td>
</tr>
<tr>
<td>(a_9) remains queued</td>
</tr>
<tr>
<td>(a_9) remains queued</td>
</tr>
</tbody>
</table>
divergence results in a single in-flow value being used to construct as many out-flow values as possible while still preserving the FIFO timing in an in-flow domain. If you want demand driven, provide the entire composition expression in one DDE. The demand divergence results in a single in-flow value being used to construct exactly one out-flow value. Demand divergence also preserves the FIFO timing in an in-flow domain. In Table 5.2 the second interpretation uses each $A$ value to construct both an $X$ value and a $Y$ value, however, each $X$ value is used to construct only one $Z$ value. We do not feel that demand driven divergence should be allowed as a simple vergence abstraction in itself. This would complicate the formal description of bubble behavior for bubbles that would receive demand driven consumable flows. Thus we allow a demand driven divergence only as an internal part of convergence flow abstraction and do not allow it to effect the way bubbles access flow values.

Non–simple strict vergences depend on the use of DDE definitions to interpret the timing of flow values. Without knowing the DDE definition of $Z$ and whether $+$ distributes over $[\ldots]$, we would not be able to know which interpretation to use. However, the balancing of non–simple vergences must only guarantee that every verging flow has a destination and every vergent flow has a source. This can be accomplished using a modified version of the Teamwork algorithm for balancing decomposition flow abstraction [29]. The algorithm given in Figure 5.36 determines whether a set of consumable flow labels $X_{set}$ balance strictly with a single consumable flow label $Z$. For convergence, the single flow label is the convergent flow and the set is the set of converging flow labels. For divergence, the single flow label is the diverging flow and the set is the set of divergent flow labels. The use of the abstract type set makes the balancing of merge and copy vergences obvious.
function ConsumeVergeBal\((Z:\text{ModifiedLabel} ; X\text{set}:\text{set of ModifiedLabel} ; DD:\text{set of DDE}):\text{boolean}\);

type
FlaggedLabel = 2\text{-tuple}(Name:ModifiedLabel,\text{Matched}:\text{boolean})

begin
let \(Z\text{set} = \{(Z, false):\text{FlaggedLabel}\}\)
loop
foreach \(x \in X\text{set}\) do
if \(\exists y \in Z\text{set} \land Name(y) = x\) then
let \(X\text{set} = X\text{set} \setminus \{x\}\)
if \(\neg\text{Matched}(y)\) then
let \(Z\text{set} = (Z\text{set} \setminus \{y\}) \cup \{(Name(y), true):\text{FlaggedLabel}\}\)
endif
endif
endfor
if \(X\text{set} = \{\}\lor \forall y \in Z\text{set}, \exists d \in DD \land EntryName(d) = Name(y) \land EntryDef(d):\text{Elementry}\) then exitloop endif
let \(NewZ\text{set} = Z\text{set}\)
foreach \(y \in Z\text{set}\) do
find \(d \in DD\) such that \(EntryName(d) = Name(y)\)
if \(EntryDef(d):\text{Composition}\) then
if \(\text{Matched}(y)\)
let \(NewZ\text{set} = NewZ\text{set} \cup \{(z, true) \mid z \in \text{ExpLabels}(EntryDef(d))\}\)
else
let \(NewZ\text{set} = (NewZ\text{set} \setminus \{y\}) \cup \{(z, false) \mid z \in \text{ExpLabels}(EntryDef(d))\}\)
endif
endif
endfor
let \(Z\text{set} = \{(z, true) \mid (z, true) \in NewZ\text{set}\}\)
\(\cup \{(z, false) \mid (z, false) \in NewZ\text{set} \land (z, true) \notin NewZ\text{set}\}\)
endloop
return \(X\text{set} = \{\}\land \forall y \in Z\text{set}, \text{Matched}(y)\)
end ConsumeVergeBal

Figure 5.36: Strict Consumable Vergence Balancing Algorithm
In all the previous examples, the composing flows \( W, X, \) and \( Y \) did not decompose before composing into \( Z \), i.e., the vergences were strict vergences. Recall a non-strict vergence allows the composing flows to decompose before composing into the composed flow. This is similar but not the same as simple vs. non-simple vergences. Simple vs. non-simple vergences apply only to consumable flows because there are several types of vergence abstractions. Non-simple vergences result from the nesting of composition operators in DDE definitions.

Strict vs. non-strict refers to whether composing flows’ DDE definitions can be applied to determine the timing of flow abstractions. For example, let us consider a third interpretation of Figure 5.29, where \( Z \) is defined as \([A+B|C]\), \( W \) is defined as \([A|C]\), \( X \) is defined as \( A+B \), and \( Y \) is defined as \([B|C]\). In this case Figure 5.29 is a non-strict, non-simple version of the cascade of vergences shown in Figure 5.37. However, Figure 5.38 could also be a representation for Figure 5.29. These two different representations of Figure 5.29 also provide for different timing of flow values. The rules for timing consumable flow vergence abstractions should make the choice of representations, and thus the timing of flow values, unambiguous.

It is possible to describe an algorithm for balancing non-strict consumable vergence abstractions. For simple bundling you would simply apply DDE definitions until all labels were defined by elementary DDEs. However, for layered bundling, the decision to apply a particular DDE changes the timing interpretation associated with that abstraction. For example, Figure 5.37 represents the case where the converging \( X \)’s DDE definition was applied. In Figure 5.38, \( X \)’s DDE definition was applied only to compose an \( X \) from subordinates of \( W \) and \( Y \). Determining the timing of these non-strict, consumable, vergence abstractions becomes a very difficult problem.
The choice of which interpretation to select becomes arbitrary simply because the traditional descriptions of flow balancing have never considered, and lend no insight into, making this choice.

Figure 5.37: Equivalent Simple Strict Vergences

Figure 5.38: Other Equivalent Simple Strict Vergences
Non-strict, consumable, vergence abstractions can also result in infinite queueing of values. For example, suppose $Z$ was defined as $A+B+C$, $X$ was defined as $A+B$, and $Y$ was defined as $B+C$. If $X$ and $Y$ converge into $Z$, then for every pair of an $X$ and a $Y$ value, a single $B$ value is left queued at the vergence point. The timing of this vergence is provided in Figure 5.39. Subsequently, a steadily growing sequence of $B$ values builds up. For this reason, we suggest that non-strict consumable vergence abstractions not be allowed.

![Figure 5.39: Timing for Growing Queue Example](image)

In the next section we use the interpretations of vergence flow abstraction to interpret decomposition flow abstraction.

### 5.4.2 Decomposition Flow Abstraction

Each DFD bubble decomposes into a child process which is either another DFD or a P-Spec. The flows into and out of a DFD bubble must be represented by flows into and out of the child process. In the case of a child DFD, the in-flows are those flows whose source is off-page and the out-flows are those flows whose destination is off-page. P-Spec flows are already organized into two sets labeled in-flows and
out-flows. As with flow vergences, the parent bubble’s in-flows and out-flows can be an abstraction of the child’s in-flows and out-flows. However, unlike flow vergences, there is no graphic indication of which child in-flows (out-flows) are represented by a particular parent in-flow (out-flow). Thus the balancing of decomposition flow abstraction must rely exclusively on DDE definitions. We can provide some help in identifying which parent flows balance with which child flows by categorizing flows by flow behavior.

Traditionally, parent in-flows can only decompose (diverge) in relationship to child in-flows and child out-flows can only compose (converge) in relationship to parent out-flows. However, since balancing is based solely on DDE definitions and flow behavior, we must balance a convergence of all parent in-flows of the same flow behavior with a divergence of all child in-flows of that flow behavior. Similarly, we must balance a convergence of all child out-flows of the same flow behavior with a divergence of all parent out-flows of that flow behavior. For consumable flows these vergences do not represent a particular type of abstraction (e.g., merge, combine, copy, split, etc.) rather just a bundling of the flows on either “side” of the decomposition abstraction; i.e., these vergences do not introduce additional timings over and above the timings inherent in the DDE definitions. For analog and persistent flows, timing is not an issue.

To balance analog (persistent) flow decomposition abstraction, perform the following steps:

1. Balance the collection of analog (persistent) parent in-flows with a convergence to a fictitious flow labeled $Y_{in}$ defined as the sum (+) of all flows in the collection.
2. Balance the fictitious flow labeled $Y_{in}$ with a divergence to the collection of analog (persistent) child in-flows.

3. Balance the collection of analog (persistent) child out-flows with a convergence to a fictitious flow labeled $Y_{out}$ defined as the sum of all flows in the collection.

4. Balance the fictitious flow labeled $Y_{out}$ with the collection of analog (persistent) parent out-flows.

To balance consumable flow decomposition abstraction, modify the algorithm given previously in Figure 5.36 so that the first argument is a set of ModifiedLabels and serves as the initial value of the variable $Zset$. To balance consumable in-flows, use the parent in-flows as the first argument and the child in-flows as the second argument. To balance consumable out-flows, use the child out-flows as the first argument and the parent out-flows as the second. The simple vergence timing diagram can be constructed by building a network of simple vergences based on the application of each DDE definition to $NewZset$ in the second loop. Note that if a label $z$ appears more than once in the composition expression associated with the DDE $d$, this represents a demand divergence. When a $Zset$ element and $Xset$ element are matched in the first loop, connect the appropriate directed edge from the $Xset$ element to the network of simple vergences on the $Zset$ side. In the assignment to $Zset$ based on the contents of $NewZset$ following the second loop, any element of $NewZset$ that does not appear in the assigned value of $Zset$ represents a copy divergence of an element previously removed from $Xset$. 
5.4.3 Other Composition Expression Operators

The flow abstractions described previously limit the syntax of DDE composition expressions. Recall we are assuming that all primitive flow labels are defined by elementary DDEs. In particular, we have required that all DDE composition expressions be constructed using only the + and [. . . ] operators. Also any flow whose label's DDE definition uses the [. . . ] operator, including subordinate DDE substitution, must be a consumable flow.

What about the other two operators a{...}b denoting iteration with optional bounds a and b and (...) denoting optional? Consider each type of flow behavior. For both analog and persistent flows, a flow label defined as \( a\{X\}b \) is inherently ambiguous because we could not distinguish between multiple values from the same domain. An analog or persistent flow label defined as optional also does not make sense. Since flows themselves are static, and for analog and persistent flows a value is always associated with a flow, saying that a value is sometimes there and sometimes not just does not work.

For consumable flows, the iteration operator might represent a transmission delay or removal of the FIFO order. For example, suppose several consumable flows all labeled \( X \) converged to a consumable flow labeled \( Z \), and \( Z \) was defined in the DD as \( 5\{X\}10 \). This might mean to nondeterministically wait until at least 5 \( X \) values, but not more than 10 \( X \) values, have arrived at the convergence point, and then to send them as a single packet on the \( Z \) flow. What happens when \( Z \) diverges into several \( X \) flows? Do we make copies of all \( X \) values and send a copy on each divergent \( X \) flow? Do we maintain the FIFO ordering of these copies? Traditional literature contains no references to iterative vergences. The convergence transmission delay
interpretation behaves like a "burst transmission" which is an implementation of a network traffic control technique. We desire our specifications to be independent of particular implementation techniques, especially something as low level as network traffic control. Thus we do not allow the iteration operator in DDE composition expressions.

The optional operator can be used to provide a prioritized demand driven divergence for consumable flows. For example, suppose consumable flows $X$ and $Y$ converged into consumable flow $Z$, and $Z$ was defined as $X+(Y)$. Figure 5.40 provides a simple vergence equivalent for this convergence. The prioritization of the demand divergence is represented by the labels high and low. When an $X$ value arrives at the demand divergence point, it chooses the high priority side if a $Y$ value is queued at the combine convergence; otherwise, it chooses the low priority side. In effect, the optional operator removes the combine queueing effect from the $X$ flow—only $Y$ flow values queue up at the convergence point. The divergence of $Z$ into flows $X$ and $Y$ is interpreted as in Figure 5.41.

![Figure 5.40: Timing of Optional Convergence](image)

If several optional operators appear in a single DDE + conjunction, as in $X+(Y)+(W)$, we convert it to a single optional form containing an alternative of all possible options, e.g., $X+([Y+W | Y | W])$. Nesting of the optional operator is represented by a sequencing of the equivalent simple vergence interpretations. Note that for domain
composition consideration $X+(Y)$ is equivalent to $[X \mid X+Y]$. As with the $[\,\mid \ldots ]$, the optional operator may only appear in the definition of consumable flow labels.

![Figure 5.41: Timing of Optional Divergence](image)

### 5.5 Flow Balancing Summarized

In this chapter we have suggested that you view the primitive flow labels as type names, modified by value names for added clarity. We have distinguished three different types of flows based on the behavior of the values they represent. Using these flow behaviors and composite flow label DD definitions, we have provided several interpretations of flow abstractions that avoid ambiguity, and in the case of consumable flows, provide for the timing and domain composition description of abstracted flow values. Although there are several choices for interpreting flow abstraction, it is more important that some choice be made rather than trying to argue that a particular choice is more correct. Making a choice provides for a precise interpretation of the meaning of a DFD with respect to the representation of values on flows.

The following Type Definitions and Invariant Properties incorporate our suggestions into the formal model of SA specifications. Our suggested interpretations of flow abstraction will be incorporated into the operational semantics presented in Chapter 9.

Our suggested balancing of flow abstraction is based on a layered bundling ab-
straction as applied to flows of a particular flow value behavior. Invariant Property 10 requires all flows into and out of the same vergence point to have the same flow value behavior.

**Invariant Property 10**

(for any $P$ of type DFD:)

$$\forall n: integer \left[ (\{FlowBehavior(f) | f \in Flows(P) \land (Source(f) \land Source(f) = n \lor Destination(f) \land Destination(f) = n)\}) \leq 1 \right]$$

In the next chapter we replace the use of composition expression operators for type definition of primitive flow labels with ADT declarations in SPECS. Thus we assumed the labels of flows which appear in P-Specs were defined in the DD by elementary definitions. Invariant Property 11 formalizes this assumption.

**Invariant Property 11**

(for any $M$ of type SA-Spec:)

$$\forall p \in Processes(M) \land p \in P-Spec \Rightarrow \forall f \in InFlows(p) \cup OutFlows(p) \Rightarrow \exists d \in DataDictionary(M) \land Entryname(d) = FlowLabel(f) \land EntryDef(d) \land Elementary$$

Our suggestions have resulted in a change to the syntax of DDE composition expressions and restrictions on the forms these expressions can take. Type Definitions 18 through 21 provide the abstract structure of our composition expressions, and replace the lexical definition of composition expressions provided by Type Definition 15 and Table 4.1 given in Chapter 4. Both conjuncts (+) and disjuncts ([...]) are described as n-ary operators on a set of composition expressions. Invariant Property
12 requires these conjuncts and disjuncts to have at least 2 operands. Expression Definition 5 from Chapter 4 is rewritten formally in Expression Definition 9. Invariant Property 13 assures that DDE substitution does not result in an infinitely recursive definition for any DDE.

**Type Definition 18**

\[
\text{Composition} = \text{Conjunct} \mid \text{Disjunct} \mid \text{Optional} \mid \text{ModifiedLabel};
\]

**Type Definition 19**

\[
\text{Conjunct} = \text{set of Composition};
\]

**Type Definition 20**

\[
\text{Disjunct} = \text{set of Composition};
\]

Composition expressions are used only to define composite flow labels. These composition expressions must not be ambiguous, and for analog and persistent flows the entire definition, including DDE substitution, cannot contain the disjunct or optional operator. Using layered bundling, ambiguity can be checked for by placing a flow label's DDE definition in disjunctive normal form (DNF). For analog and persistent flows, the DNF should contain only one conjunct. To check for the use of disjunct or optional operators in the entire definition of an analog or persistent flow label, we place the entire definition, including DDE substitution, in DNF and check that the DNF contains only one conjunct. Figure 5.42 provides an algorithm for deriving the DNF of the definition of a flow label. The second parameter of the algorithm toggles
Type Definition 21
Optional = 1-tuple(Expr:Composition);

Invariant Property 12
(for any $E$ of type Conjunct or type Disjunct:)
$|E| \geq 2$

Expression Definition 9
define $ExpLabels(C:\text{Composition})$ as set of ModifiedLabel such that

$$C:\text{Conjunct} \lor C:\text{Disjunct} \Rightarrow ExpLabels = \left( \bigcup_{c \in C} ExpLabels(c) \right)$$

$\land C:\text{Optional} \Rightarrow ExpLabels = ExpLabels(Expr(C))$

$\land C:\text{ModifiedLabel} \Rightarrow ExpLabels = \{C\}$

Invariant Property 13
(for any $M$ of type SA-Spec:)

$$\forall d \in DataDictionary(M) \land EntryDef(d):\text{Composition} \Rightarrow EntryName(d) \notin \left( \bigcup_{l \in ExpLabels(EntryDef(d))} AllLabels(l, DataDictionary(M)) \right)$$
the inclusion of DDE substitution in the derivation. Invariant Property 14 assures
that composite flow labels do not have ambiguous DD definitions, and that all flow
values have a non-empty domain composition; i.e., each conjunct must have at least
one non-optional operand. Invariant Property 15 assures that analog and persistent
flow label DD definitions do not use disjunct or optional operators.

**Invariant Property 14**

(for any $M$ of type SA-Spec:)
\[ \forall d \in \text{DataDictionary}(M) \land \text{EntryDef}(d), \exists \text{Composition} \]
\[ \Rightarrow \neg \text{Ambiguous}(\text{DNF(EntryDef}(d), \{\})) \land \\
\{\} \notin \text{Structure}(\text{DNF(EntryDef}(d), \{\})) \]

**Invariant Property 15**

(for any $M$ of type SA-Spec:)
\[ \forall d \in \text{DataDictionary}(M) \land \text{EntryDef}(d), \exists \text{Composition} \land \\
\exists p [p \in \text{Processes}(M) \land \exists \text{FD} \land \\
\exists f [f \in \text{Flows}(p) \land \text{FlowLabel}(f) = \text{EntryName}(d) \land \\
(\text{FlowBehavior}(f) = \text{analog} \lor \text{FlowBehavior}(f) = \text{persistent})] ] \\
\Rightarrow |\text{Structure}(\text{DNF(EntryDef}(d), \text{DataDictionary}(M)))| = 1 \]

The balancing of vergence and decomposition abstractions are not asserted by
invariant properties. The methods for balancing flow abstractions given in this chap­
ter have sufficient detail that these invariant properties can be easily inferred. Flow
abstraction balancing is not traditionally considered a syntactic attribute of an SA
specification, but rather is a semantic issue that needs to be checked for. This can
also be said for some of the issues we did formalize by invariant properties; however,
it is more true for balancing.

Flow abstraction can be a useful tool for uncluttering DFDs. However, we must
function $DNF(C:\text{Composition}; DD:\text{set of DDE}):\text{DNFType}$;
where \(\text{DNFType} = 2\text{-tuple(}\text{Structure} : \text{set of LabelConjunct},\text{Ambiguous} : \text{boolean})\);

\[\text{LabelConjunct} = \text{set of ModifiedLabel};\]
begin
if $C$?ModifiedLabel then
if \(\exists d \in DD \land \text{EntryName}(d) = C \land \text{EntryDef}(d) : \text{Composition}\) then
return $DNF(\text{EntryDef}(d), DD)$
else return \{ \{ C \} \}, false$:$\text{DNFType}
endif
elseif $C$?Optional then
let $d = DNF(\text{Expr}(C), DD)$
return \{ \{ \} \} $\cup \text{Structure}(d), \text{Ambiguous}(d))$\:$:\text{DNFType}$
elseif $C$?Disjunct then
let $D = \{ DNF(c, DD) \mid c \in C \}$
return $\left( \bigcup_{d \in D} \text{Structured}(d) \right), \left( \bigvee_{d \in D} \text{Ambiguous}(d) \right)$:$\text{DNFType}$
else (* $C$?Conjunct *)
let $D = \{ DNF(c, DD) \mid c \in C \}$
let $ambig = \left( \bigvee_{d \in D} \text{Ambiguous}(d) \right)$
pick $x \in D$
let $D = D - \{ x \}$
let $result = \text{Structure}(x)$
while $D \neq \{ \}$ do
pick $y \in D$
let $D = D - \{ y \}$
if \(\exists v \in result \land \exists u \in \text{Structure}(y) \land v \cap u \neq \{ \}\) then
let $ambig = true$
endif
let $result = \{ v \cup u \mid v \in result \land u \in \text{Structure}(y) \}$
endwhile
return $(result, ambig):\text{DNFType}$
endif
end $DNF$

Figure 5.42: Disjunctive Normal Form Algorithm
be careful to not abstract too much or we will lose the expressiveness of graphical
techniques like DFDs by burying our specification details in the DDE definitions. In
the next chapter we compare the traditional specification of data structure with the
techniques available in the specification language SPECS.
CHAPTER 6. SYNTHESIS OF STRUCTURED ANALYSIS AND
OBJECT-ORIENTED SPECIFICATIONS

6.1 Overview

In Chapter 5 we assumed the label of any P-Spec flow was defined by an elementary DDE. This allowed us to focus on the use of composition expression operators for representing flow abstraction. However, our assumption does not characterize the usual traditional DD definition of P-Spec flow labels. In this chapter we develop an object-oriented characterization of flow value structures. Thus we embed object-oriented specifications directly in an SA specification. This differs from Bailin’s approach [5]. Bailin uses SA specifications to derive an object-oriented design. We do not replace SA functional decomposition, but rather augment the specification of P-Specs by providing higher order abstract objects in which to couch P-Spec functionality.

P-Specs represent the primitive functional unit of an SA specification. Each P-Spec describes a mapping of in-flow values to out-flow values. This mapping is traditionally described in a high level pseudo-code. The P-Spec must be able to access the individual subordinates of each in-flow value and be able to define the individual subordinates of each out-flow value. Thus a P-Spec must know the structure of these flow values. The DDE composition expression is the traditional method of specifying
this structure. This represents a dual usage for composition expressions. Composition expressions are being used both to define flow abstraction and to describe flow value structure.

Traditionally, there is not a clear separation of these two usages. Depending on the type of composition operators used, a non-elementary DDE may be defining both flow abstraction and flow value structure. For example, consider the DFD and associated P-Specs shown in Figure 6.1. Both Bubble1 and Bubble2 are representing P-Spec processes. Assume $Z$ is defined in the DD as $Y + X$. The flow $Z$ is shown diverging into a copy of itself and a copy of its $Y$ subordinate. Assume $Z$ appears as an in-flow in P-Spec1 and $Y$ appears as an in-flow in P-Spec2. The composition expression operator $+$ has been used to describe both the flow abstraction in the DFD and the flow value structure of the $Z$ in-flow in P-Spec1. The traditional method of referencing the two subordinates of the $Z$ in-flow in P-Spec1 is to refer to $Z.X$ and $Z.Y$ in the pseudo-code.

Using the traditional interpretation of flow value structure we can abstract the $Z$ in-flow into two separate in-flows, $X$ and $Y$, in P-Spec1. The pseudo-code references then become just $X$ and $Y$. Thus we can separate flow abstraction use and flow value structure use of composition expression operators. This allows us to investigate the potential for objected-oriented specification of flow value structure.

In this chapter we examine the traditional definition of flow value structure using DDE composition expressions. We then compare composition expression use for defining flow value structure with the facilities of SPECS for defining flow value structure. We complete this chapter with our suggestions for using SPECS to define flow value structure and a general discussion of our views on the merging of SA
specifications and object-oriented specifications.

6.2 Traditional SA Data Structuring

The use of the composition expression for defining flow value structure originated with De Marco [18]. De Marco's original work did not allow for DFD vergences, but did allow for decomposition flow abstraction. De Marco calls this parallel decomposition of data and function. De Marco's composition expression operators originated with Bohm and Jacopini's claim that any process or program can be made of elements which are related in three ways:

1. a sequential set of instructions,

2. a closed-end decision construct (e.g., if-then), or
3. a closed-end looping construct (e.g. while-loop).

De Marco states that these same three relationships will also serve to describe data. He rephrases them as:

1. Sequence: the concatenation of two or more subordinates in order. (The + operator.)

2. Selection: the choice of precisely one of two or more alternatives. (The \[. . .\] operator.)

3. Iteration: the repetition of a designated subordinate zero or more times. (The \[a\{ \}b\] operator with optional bounds \(a\) and \(b\).

De Marco's definition is clearly a lexical definition of a flow value. Concatenation, selection, and iteration are exactly those operators which appear in BNF grammars.

Gane and Sarson recognized subordinates of a flow value as being mandatory, an alternative, optional, or iterated one or more times [22]. These can be characterized using De Marco's notation. Any flow value is composed of a concatenation of its subordinates (although it is not clear that an order is intended). The default is that each subordinate is mandatory. An alternative subordinate is specified using De Marco's selection operator. Optional is iteration zero or one time. In fact De Marco uses parentheses as a short hand for iteration zero or one time, i.e. optional. Gane and Sarson's iterated operator is equivalent to De Marco's iteration operator with a lower bound of one.

Yourdon uses the same syntax and semantics as De Marco, but does not refer to an order within a concatenation [66]. Yourdon refers to the concatenation operator
as meaning and. A flow value structure of $X + Y$, means that any flow value is composed of both an $X$ value and a $Y$ value. Lexical order does not matter. Consider the example of Customer = Lastname + Firstname. Alphabetically the Lastname is usually taken first, yet when written down or spoken the Firstname usually is used first.

Yourdon and Ward introduced DFD vergence flow abstraction and interpreted them similar to De Marco's decomposition flow abstraction [59, 66]. This further removed the lexical concatenation semantics from the + operator. As discussed in Chapter 5, graphically the + operator is both associative and commutative. Modern structured analysis views the composition expression not as a definition of lexical structure but rather as a definition of type structure. What exactly is the type structured interpretation and how does it compare to the abstract modeling abilities of SPECS?

6.3 Comparing Composition Expressions and SPECS

Abstract model specifications are well-defined mathematically and come with a rich collection of primitive types, structured types, and built-in operations. How do abstract model type declarations compare with the composition expressions commonly used in SA? Are there benefits to using one or the other to represent the type structure of DDEs?

6.3.1 Structured Types

De Marco's concatenation operator is a structuring mechanism for combining a finite number of potentially heterogeneous subordinates. This can be seen as similar
to a cross product of several domains. Each domain is represented by a modified label which is defined in another DDE (increasing the size of the DD), a literal, or a nested composition expression. Note that a nested composition expression has no name for the domain it represents, and that a literal represents a domain of size one. The only operation defined on concatenation structures is dereferencing, and this only works if the arguments in the expression are labels, not nested composition expressions or literals. Dereferencing syntax is a dot ("." ) followed by the subordinate name.

As with flow labels, the question of how to interpret these domain names (if present) arises. Do we interpret the name as a value (i.e. variable) name or as a type name? With composite flow labels, we could view them as both because the flow values are explicit abstractions of primitive flow values and are never directly referenced by a P-Spec. The primitive flow labels are then treated as variables in P-Specs to facilitate access their flow values. However, if we view subordinate names of these primitive values as both value names and type names, then we end up with the same concern as before: to avoid ambiguity in referencing values we create the necessity of redundant definitions of type structure. For example, if the previous example of \( \text{Customer} = \text{Lastname} + \text{Firstname} \) is replaced by \( \text{Customer} = \text{Name} + \text{Name} \), where \( \text{Name} \) is defined in the DD as a string of letters, then we can’t necessarily determine which \( \text{Name} \) field represents the Firstname and which the Lastname. However, using the original representation of \( \text{Customer} \) would result in redundant DDEs for \( \text{Lastname} \) and \( \text{Firstname} \).

This same structuring mechanism is provided by the tuple structure of abstract model types. The only operation defined on tuples is also dereferencing. However, tuple components are separately named and typed. The names are used in a func-
tional notation for dereferencing. The types need only be further defined if they are non-primitive types. The Customer example becomes,

Customer = 2-tuple(Lastname : string,
                     Firstname : string);

Furthermore, tuple constants can be constructed without having to name them with a variable. For example, '("Smith","John")':Customer' defines a tuple constant of type Customer. There does not currently exist a syntax for defining a constant in the domain of Lastname + Firstname without providing a variable of type Customer to store that value.

De Marco's selection operator provides an alternative typing mechanism. Again the only standard operation is dereferencing with the same restrictions and syntax as concatenation. This alone creates confusion. The syntax for concatenation and selection dereferencing is exactly the same. Alternatives containing modified labels require additional DDEs as with concatenation. Additionally, the P-Spec writer must determine which alternative is valid. This traditionally is done in pseudo-code by saying "if Z is an X then ...", etc., where Z is defined in the DD as [X | ...]. If X is a literal, then we can write "if Z = X then ..." etc.

Abstract model types also have an alternative structured type. Each type alternative is required to be uniquely named. Dereferencing is automatic by means of the is-of-type operator (?). For example, if Z was defined as real | string, we might write the following code fragment:

if Z:=real then
    Z← 10.3
De Marco's iteration operator provides a structuring mechanism for combining a collection of homogeneous subordinates. Again, the only operation is dereferencing with the same restriction, syntax, effects on DD size, and dereferencing problems as concatenation and selection. However, it is unclear which of the homogeneous subordinates is referred to by the dereferencing operation. Is it the first, in which case we are implying an order to the subordinates? Is it non-deterministic?

Abstract model types provide two mechanisms for combining a collection of homogeneous components: set and sequence. Although a sequence can be modeled as a set of ordered pairs, with one component serving to define the order, sequence provides a well-defined abstraction which avoids the underlying details of ordering set elements. Both set and sequence come with a wealth of operations. Deterministic dereferencing is possible with sequence, but not set, because of the implied order. Set dereferencing is accomplished by quantification of variables and the is-an-element operator (∈). Sequences can be dereferenced by the index function.

De Marco's optional operator has no particular syntax for dereferencing. The optional operator is simply a short hand for an iteration of zero or one time. This can be expressed in SPECS in a straightforward approach based on iteration using a set or sequence with a bounded size of one. Another SPECS representation, which

---

1The ∈ operator is used for sequence membership, but does not determine position in the order.
we feel is more appropriate, is based on our disjunctive normal form for domain composition used to balance consumable flow abstraction. Recall that $X + (Y)$ can be interpreted as $[X \mid X + Y]$. Thus we could use a SPECS alternative type of two types, one representing $Y$ and the other representing the absence of $Y$. Absence can be defined using the notion of a zero tuple, i.e., no information. The entire translation of $Z$ defined as $X + (Y)$ into SPECS is as follows:

$$Z_{\text{Type}} = 2\text{-tuple}(X;X_{\text{Type}}, Y;\text{MaybeY}_{\text{Type}});$$

$$\text{MaybeY}_{\text{Type}} = Y_{\text{Type}} \mid \text{NoY}_{\text{Type}};$$

$$\text{NoY}_{\text{Type}} = 0\text{-tuple}();$$

Clearly the SPECS type structuring facilities are superior to composition expressions. SPECS provides separation of type name from value name, a mathematical foundation, and a large collection of built-in operations on each structured type. Besides structured types, SPECS also provides a rich collection of primitive types, e.g. integer, real, char, etc. What facilities are there in composition expressions for providing primitive domain values?

6.3.2 Primitive Types

Recall the two types of DDE definitions, elementary and composition. Elementary entries are said to be "self-defining." Composition entries are defined by a composition expression. The only primitive domain in the composition expression syntax is the literal. Each literal defines a domain of size one. Using the composition
expression operators literals can be combined to form more interesting domains. The most common of these is a selection of several literals. For example, Digit might be defined as [ "0" | "1" | "2" | ... | "9" ]. Elementary entries and entries defined in terms of literals (i.e., quoted strings) are the primitive pieces out of which composition entries are composed. Elementary entries and literals serve the same role that primitive types, e.g. INTEGER, REAL, CHAR, etc., serve in structured type definitions. However, elementary entries have no formal definitions. They are "self-defining."

De Marco defines two types of primitive DDE domains: discrete and continuous [18]. A discrete entry represents some finite set of value choices. A continuous entry represents some range of value choices. Gane and Sarson note that it is sometimes difficult to draw the line between what is discrete and what is continuous [22]. For example, is the range of integers from 1 to 100 discrete or continuous? If we use "finite set" to characterize discrete, then any range of digital values on a finite machine is discrete. In this case the only true continuous values are analog signals. Would we then want to think of a range of floating point numbers as discrete? Alternatively, if our range of integers from 1 to 100 is continuous then at what set size do we call a set of values discrete? When a discrete entry represents a small set of potential values and these values can be defined as literals then an alternative composition expression of these literals can be used to define the discrete entry.

The issue of discrete verses continuous is further clouded by other uses of these terms. In Chapter 5 we examined two other uses of the terms discrete and continuous. We used Ward's discrete-existence/continuous-existence and Hatley's discrete-change/continuous-change to derive our three flow value behaviors [27, 59]. We also noted that analog flow labels should define a continuous domain.
Consider the original syntax of composition expressions presented in Chapter 4. In a sense, composition expressions are like a fire escape on a burning building. When you get to the third floor—the elementary entries—there's no fire escape left, and you have to jump. If you define everything down to the ground using literals, then you need to define the literal values associated with every primitive domain.

Consider the DD definition of a primitive domain integer in Table 6.1. (Assume integer does not allow leading zeros.) If we had wanted a subrange of integers, say 1 to 100 inclusive, we would need a more elaborate definition as in Table 6.2. Consider extending this definition style to reals and strings. (This would be analogous to a fire escape with so many tiny steps, the building would completely burn down before you made it to the ground, but you wouldn't want to skip any steps or your definitions would be incomplete or ambiguous.) What is needed is a formalism that defines primitive domains without requiring lexical descriptions of all possible values.

Table 6.1: Syntactic Definition of integer

<table>
<thead>
<tr>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer = non-zero-digit + {digit}</td>
</tr>
<tr>
<td>non-zero-digit = [ &quot;1&quot;</td>
</tr>
<tr>
<td>digit = [ &quot;0&quot;</td>
</tr>
</tbody>
</table>

Table 6.2: Integer Subrange Definition

<table>
<thead>
<tr>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer = [ &quot;100&quot;</td>
</tr>
<tr>
<td>non-zero-digit = [ &quot;1&quot;</td>
</tr>
<tr>
<td>digit = [ &quot;0&quot;</td>
</tr>
</tbody>
</table>

What is required is a formal method of specifying the basic building blocks of type definitions. In abstract model specifications, the primitive types are chosen as commonly accepted mathematical types, e.g., mathematical integers, mathematical
reals, characters, strings, etc. These domains have intuitive and well-defined semantics and come with a host of built-in operations. Clearly abstract model primitive types provide a more solid foundation for defining domains than the existing definitions of primitive DDE domains. Integers, reals, booleans, character sets, and strings all provide an abstraction of values that do not need to be defined lexically. Subranges and other restrictions on these domains can be specified by invariant properties using first order logic. User-defined types (as in Pascal) provide the same kind of primitive type as an alternative composition of literal values. For example, the primitive DDE \texttt{color} = [ "red" | "green" | "blue" ] can be replaced by \texttt{color} = (red,green,blue). In the next Section, we present our suggestions for using SPECS ADT to specify the type structure of primitive flow values.

6.4 Using SPECS ADTs

Clearly SPECS abstract model types provide a more useful perspective on flow value structure than the lexical structure defined using composition expressions. Abstract model types, both primitive and structured, can be used to define type structure in place of the traditional elementary DDEs and composition expressions. Abstract model types have the additional benefits of well-defined primitives and a wealth of built-in operations on both primitive and structured types. The built-in operations defined for SPECS primitive and structured types provide a useful framework in which to couch the description of P-Spec behavior. Abstract model specifications facilitate the writing of both assertional and algorithmic P-Specs. In Chapter 7, we address the formalization of P-Specs using the underlying mathematical formalisms of abstract model operations.
Abstract model invariants can be used to restrict domain values rather than taking a BNF approach to describing potential values. Furthermore, the abstract model approach allows the construction of abstract operations built on top of the built-in primitive operations. This incorporates the accepted object-oriented concepts of data abstraction, encapsulation, and information hiding within the scope of SA specifications.\(^2\) P-Specs serving as the source of a flow will only be allowed to perform create and modify operations, and P-Specs serving as the destination of a flow will only be allowed to perform query operations.

With this perspective on flows, each primitive analog flow and each primitive persistent flow is essentially an instance of an ADT. Each primitive consumable flow value is also an instance of an ADT. The ADT specification is provided by an elementary DDE associated with a primitive flow's label. Using ADTs to define primitive flow value type structures instead of composition expressions decreases the size of the DD because subordinate labels appearing in composition expressions must be defined by a DDE.

Recall that several different flow labels may share the same DDE if they differ only by the modifier component of the flow label. Usually modifiers serve only to distinguish between two different flows with the same destination and structure. However, it may be that certain modifiers represent subtypes of the ADT defined by an elementary DDE. Each flow value associated with a particular DDE must have the type structure defined by the ADT—this includes satisfying any invariant properties. We can then extend the SPECS ADT syntax to allow additional invariant properties.

\(^2\)The functional decomposition provided by the DFD hierarchy also provides a form of abstraction and functional information hiding.
associated with particular modifiers.

6.5 Merging Objected-Oriented Specifications and SA

We have shown the ability and benefits of using the abstract model approach (SPECS) for specification of elementary DDEs—those DDEs associated with primitive flows. In the remainder of this thesis we assume the use of the DDE composition operators for type structuring have been replaced with the abstract model approach for specifying ADTs. We now modify Type Definition 14, so that an elementary DDE is defined by an abstract model ADT specification. Type Definition 14 is replaced Type Definition 22. The BNF for an ADT is not given, but rather can be inferred from the contents of Chapter 3. This BNF should be modified to allow specification of subtypes by invariant properties on modified labels. The semantics of SPECS ADTs are as defined in Chapter 3 and are not formalized further.

Type Definition 22
Elementary = BNF[ADT];

To support abstract type composition we allow ADT specifications to refer to other ADTs defined by elementary DDEs. Thus there may exist elementary DDEs that are not associated with any primitive flows, but just serve to define an ADT used in the composition of another ADT. Still ignoring DDEs associated with stores, DDEs are divided into two sets. Non-elementary DDEs are defined by a composition expression. All non-elementary DDEs are associated with composite flow labels and represent flow abstraction. Elementary DDEs are defined by a SPECS ADT. Each primitive flow label is defined by an elementary DDE.
Our DD has now become a depository for flow abstraction representations (i.e., composition expressions) and object-oriented specifications (i.e., SPECS ADTs). The DFD hierarchy provides a functional decomposition of the entire system. The primitive flows provide objected-oriented specifications of objects computed over by P-Specs. Thus the primitive flows represent a boundary between the top-down functional decomposition provided by the DFDs and the bottom-up composition of objects provided by ADT composition. (See Figure 6.2.)

![DFD Hierarchy](image.png)

**Figure 6.2: The DFD/ADT Boundary**

It is our contention that as the database of abstract model specifications grows, this boundary will appear closer to the context DFD. However, for many types of systems, the DFD hierarchy will not disappear altogether as many objected-oriented purist advocate. Although objected-oriented specifications are a powerful tool for the
development of systems, it is our belief that the construction of a "top-level" object
that puts all the communication of sub-objects together is often artificial. Deciding
which object a particular operation should belong to is sometimes arbitrary and it is
these operations that really represent the functional decomposition provided by the
DFD hierarchy. This may be especially true for embedded real-time systems. In the
next chapter we look inside this boundary between methodologies to see how SPECS
ADTs can help facilitate the formalization of P-Specs.
CHAPTER 7. SPECIFYING PROCESS BUBBLES

7.1 Overview

DFD bubbles represent the procedural component of traditional SA. DFD bubbles have been called processes, data transactions, functions, and when applied to bubbles representing P-Specs, functional primitives. The role of a DFD bubble is to transform in-flow values into out-flow values.

In this chapter we develop a state based assertional specification of DFD bubbles. This state based assertional specification will facilitate the representation of DFD bubble execution similar to the execution of a Petri net [51]. Satisfaction of an assertion will determine if the bubble is enabled. The resulting "marking" of flow values on flows will also be specified by an assertion.

In Section 7.2, we discuss the traditional methods and extensions for the specification of a DFD bubble. One of these extensions, Kung's information flow expressions [35, 36], provides the foundation for our development of a modified state transition diagram specification of DFD bubble behavior. Kung's information flow expressions provide a specification of flow value consumption (read or receive) and flow value production (write or send). Kung's work does not consider the specification of the functional value of flows, but rather just the timing of flow value consumption and production. In Section 7.3, we present Kung's initial work and demonstrate a fun-
damental deficiency of information flow expressions for the complete specification of bubble timing. We then present an augmented version of Kung's information flow expressions which corrects this deficiency. In Section 7.4, we present a specification method which combines the semantics of the augmented information flow expressions with first order predicate calculus to provide for both the timing specification and functional specification of DFD bubbles.

7.2 What is a Process Bubble?

The specification of a DFD bubble is provided by the process it represents—either another DFD or a P-Spec. Recall that P-Specs are traditionally specified textually by structured English, decision tables, or decision trees [18]. Yourdon adds to this list the use of pre/post conditions [66]. Yourdon allows each P-Spec to have any number of pre/post condition pairs. However, Yourdon provides two alternative semantics for his preconditions.

1. The precondition represents a guarantee from the user that the precondition will be true when the process executes. However, in the case of multiple preconditions it is unclear what happens if more than one precondition is true.

2. The process is a “sleeping princess” and the precondition represents a “magic kiss” which awakens the process and sets it to work. In this case, we assume that a non-deterministic choice is made if more than one precondition is true.

These two alternatives bring up our central question about process bubbles. Do they represent traditional imperative procedures or distributed/concurrent processes? The first alternative provides the traditional semantics of preconditions in an
imperative procedure. However, imperative procedures are usually allowed only one precondition. The first alternative also does not provide a specification of how the P-Spec begins executing. There is only a statement of what is true when it does begin executing. This is because traditionally all P-Specs were designed into procedures in the resulting software product. Subroutine transfer of control was the mechanism for executing the procedural implementation of a P-Spec. P-Specs themselves were not executionally interpreted.

The second alternative does provide a method for specifying how the P-Spec begins executing. The precondition itself is serving as an enabling rule. When the precondition is true, the P-Spec is ready to execute and does.

What exactly is the distinction between an imperative procedure and a distributed/concurrent process? Although the distinction may be language dependent, we feeling the following statements represent the traditional interpretations of imperative procedures and distributed/concurrent processes. A procedure

- is not always active (does not have self-control),
- does not decide when to execute,
- when executed is usually provided with the same set of input variables (formal parameters) and produces the same set of output variables, and
- does not remember (locally) results from previous executions.

There may be special language features in some imperative programming languages which circumvent the last two items (e.g. non-traditional parameter passage mechanisms and local permanent storage). On the other hand, a process
• once initially activated, remains active,

• decides for itself when to execute (usually by monitoring external events),

• when executed might not require all input variables and might not define all output variables, and

• can remember (locally) results from previous executions.

This self-control can be characterized as representing an active entity rather than a passive entity like a procedure. The remembrance of previous results is part of this activeness. Because a distributed/concurrent process is always active, local storage is not reclaimed when it finishes an execution cycle.1

Wards “real-time” extension of SA specifications provides for the direct specification of when a DFD bubble can execute [59]. Ward provides for two types of DFD bubbles. One is the traditional DFD bubble which he calls a data transformation. The other is called a control transformation and is said to represent timing.2 Control transformations serve to activate and deactivate data transformations through activation and deactivation flows. Data transformations report event happenings to control transformations through signal flows. Activation, deactivation, and signal flows represent what Ward calls control flows. Control flow values do not contain information themselves but rather their presence reflects some implicit knowledge. A control flow domain thus has size one and can be modeled by a 0-tuple. Control transformations may only send and receive control flows. Control transformations,

1 An execution cycle represents one execution of the process algorithm.

2 Real-time specification timing, as defined by Ward, does not provide for specification of execution duration, but rather for the specification of when conditions allow execution of DFD bubbles.
once initially activated, remain active unless deactivated by a control transformation higher up in the DFD hierarchy. Control transformations are specified by a Mealy-type state transition diagram (STD). Each signal from a data transformation or activation/deactivation from an off-page flow received by a control transaction causes a state change and possibly output of control flow values.

Data transformations are active only when activated by a control transformation. They may also use the presence of an active input to decide when to execute. An active input is defined as an in-flow value that arrives independent of any action by the receiving data transformation. Thus a consumable flow value is an active input, messages arrive because of the sending data transformation. Analog and persistent flow values are not active, they must be read by the receiving data transformation. In Ward's transformation schema, each data transformation may have at most one active input flow but may have several active output flows [59]. An active output represents the sending of a value on a consumable flow. A data transformation with no active input flows executes once for each activation flow value it receives. The activating control transformation must assure that the data transformations input values exist prior to activation. A data transformation with active inputs must first be activated by an activation flow value and then executes once for each active input received until deactivated by a deactivation flow value. In either case a data transformation may produce at most one active output. Thus multiple active output flows represent an alternative of results.

Ward's transformations exhibit behaviors of both imperative procedures and distributed/concurrent processes. Since at most one active input flow is allowed for a data transformation, all other flows must be analog or persistent. Analog and persis-
tent flows always have a defined value. Since the data transformation must wait for an active input before executing, all input flows (variables) must be defined. However, at most one active output is produced. Thus not all out-flows are necessarily defined. The major differences between data transformations with and without active input flows is when the transformation is allowed to execute. Data transformations with an active input flow are more process-like than data transformations that execute just once for each activation, because they have some independence from the controlling transformation on the number of execution cycles. As long as they continue to receive active inputs and are not deactivated by a control transformation, they continue to execute.

Typically, only one control transformation may have activation and deactivation flows into a particular data transformation. These data transformations must also appear in the same DFD as the control transformation. In effect, the control transformation DFD hierarchy provides a representation of control structure similar to the imperative structure chart (i.e. calling chart) for imperative programs.

Kung's and our work lead us to question whether control transformations are even necessary [36, 15]. Control transformations impart an implementation strategy for controlling the execution of the functional data transformations. Kung's and our work provide for the execution of data transformations (DFD bubbles) by self-determination.

Kung provides an alternative view for the activation and deactivation of DFD bubbles [36]. Kung identifies a significant deficiency with DFD bubbles. Suppose a DFD bubble B has 4 consumable in-flows and 2 consumable out-flows. Also suppose that exactly two of the in-flow values are sufficient for B to transform these in-
flow values into one of the out-flow values; and that the other two in-flow values are sufficient for B to produce the other out-flow value. Traditional SA methods do provide very limited notations for describing "which in-flows" are required to produce "which out-flows." De Marco allows for the use of * and © to represent that a bubble requires or produces a conjunction or disjunction, respectively, of flow values in a single execution [18]. However, De Marco's "procedural notation" does not allow for the nesting of these operators. De Marco says "the use of procedural annotation ought to be discouraged", "try not to use it at all or use it only sparingly". He goes on to say "procedural characteristics are, after all, the very kinds of things that we can most safely defer (until implementation)." Ward's schema would require the bubbles that produced the in-flows to signal a control transformation which would then activate the bubble which required those values [59]. Note Ward's schema does not allow multiple consumable in-flows.

To represent the enabling of execution, Kung introduces information flow expressions, and a graphic representation of information flow expressions, to allow for more precise specification of which combinations of in-flow values are required by a DFD bubble, and which out-flow values will be produced [36]. Kung's information flow expressions are an enhanced nestable version of De Marco's * and © operators. We view Kung's information flow expressions as boolean expressions defined in terms of the in-flow and out-flow labels. Kung also defines a technique for using information flow expressions on bubbles which decompose into another DFD to verify that the parent bubble's specified behavior is consistent with the specified behavior given for each of the bubbles appearing in the child DFD. However in the next Section, we will

^Kung's term information flow is equivalent to our term flow.
show that Kung's technique fails because it does not take into account remembrance of past executions, which we call persistent state.

7.3 Kung's Process Specifications

In this Section we identify and correct two problems with Kung's work on information flow expressions. The first problem involves a synchronization error in the specification of the traditional producer-consumer problem, which Kung uses as his primary example [36]. This synchronization problem is inherent in the producer-consumer system.

The second problem is more fundamental. It involves a structural deficiency in information flow expressions. We show that this structural deficiency precludes arguing decomposition consistency as proposed by Kung.

In Section 7.3.1 we repeat Kung's information flow expression definitions. In Section 7.3.2 we replicate Kung's producer-consumer example, identify the synchronization problem, and provide a correction for the problem. The structural deficiency of information flow expressions is described in Section 7.3.3. The direct and significant effect of this deficiency on the verification of decomposition consistency is carefully described. We propose another perspective on the structure of information flow expressions and in Section 7.3.4 demonstrate that this alternative view corrects the problems of bubble decomposition consistency checking.

7.3.1 Information Flow Expressions

There are two possible components of the behavior of a DFD bubble. Kung identifies the dynamic (timing) component, which describes "which in-flows" are
required to produce "which out-flows" [36]. The functional component, which is missing in earlier work, specifies the out-flow values in terms of the in-flow values.

Kung uses two information flow expressions to try to specify the dynamic component of a DFD bubble: an input interface and an output interface. We can view an input interface as a logical expression defining the possible combinations of in-flow values required for a single execution of the DFD bubble. For analog and persistent flows required is equivalent to the flow value will be read and for consumable flows required is equivalent to a value is present on the flow, will be removed, and used in the execution. Likewise, an output interface defines the possible combinations of out-flow values produced by a single execution of the DFD bubble. For analog and persistent flows produced is equivalent to a flow value was written and for consumable flows produced is equivalent to a flow value was sent. Given the existence of in-flow values that satisfy the input interface, the DFD bubble executes and generates out-flow values that satisfy the output interface.

Kung uses three operators to construct information flow expressions. They are conjunction (•), disjunction (o), and exclusive disjunction (⊕). Kung provides the following syntactic definition of information flow expressions.

**Definition 3**

An information flow expression (IFE) is recursively defined as follows:

1. an information flow label is a IFE;
2. if α is an IFE then (•α), (oα), and (⊕α) are IFE’s;
3. if α and β are IFE’s then (α•β), (αoβ), and (α⊕β) are IFE’s; and
4. a string of symbols is an IFE if and only if it can be obtained by finitely many applications of the rules 1, 2, and 3.

Parentheses may be omitted and precedence is in decreasing order as follows: any unary operator, conjunction, exclusive-disjunction, disjunction. Kung refers to an IFE as a simple expression if it does not contain any operators. An IFE with operators is called a compound expression.

Kung defines two special information flow expressions, an input interface and an output interface, as follows.

**Definition 4**

The input (output) interface of a process bubble P is an IFE comprising all the labels of in-flows to (out-flows from) P.

The following two definitions define Kung’s semantics of the operators in output and input interfaces. These semantic definitions define what it means for a collection of out-flow (in-flow) values to satisfy an output (input) interface.

**Definition 5**

Let $\beta_1$ and $\beta_2$ be two information flow expressions comprised only of labels of out-flows from a process bubble P. We say that

- $\beta_1$ and $\beta_2$ are in conjunction w.r.t. (with respect to) P, denoted $\beta_1 \cdot \beta_2$, if the execution of P produces flow values satisfying both $\beta_1$ and $\beta_2$. $\beta_1$ is in unary conjunction w.r.t. P, denoted $\bullet \beta_1$, if the execution of P necessarily produces flow values satisfying $\beta_1$. 
• $\beta_1$ and $\beta_2$ are in disjunction w.r.t. $P$, denoted $\beta_1 \circ \beta_2$, if the execution of $P$ produces flow values satisfying either $\beta_1$ or $\beta_2$ (or both). $\beta_1$ is in unary disjunction w.r.t. $P$, denoted $\beta_1 \circ \beta_2$, if $P$ possibly produces flow values satisfying $\beta_1$.

• $\beta_1$ and $\beta_2$ are in exclusive disjunction w.r.t. $P$, denoted $\beta_1 \oplus \beta_2$, if the execution of $P$ produces flow values satisfying either $\beta_1$ or $\beta_2$ but not both. $\beta_1$ is in unary exclusive disjunction w.r.t. $P$, denoted $\oplus \beta_1$, if the execution of $P$ does not produce flow values satisfying $\beta_1$.

From Kung's semantic definitions we can interpret an arbitrarily nested output interface as a logical expression defining what is produced by an execution of $P$, i.e., after an execution of $P$ the output interface is guaranteed to be true. The label of an out-flow is true when a flow value is produced. The binary output interface operators have the standard semantics of logical conjunction, disjunction, and exclusive disjunction. The unary output interface operator for exclusive disjunction actually has the semantics of boolean negation. Unary disjunction just represents the constant function true, and unary conjunction just represents the identity function.

Definition 6
Let $\alpha_1$ and $\alpha_2$ be two information flow expressions comprised only of labels of in-flows to a process bubble $P$. We say that

• $\alpha_1$ and $\alpha_2$ are in conjunction w.r.t. $P$, denoted $\alpha_1 \bullet \alpha_2$, if $P$ requires flow values satisfying both $\alpha_1$ and $\alpha_2$ to produce output. $\alpha_1$ is in unary conjunction w.r.t. $P$, denoted $\bullet \alpha_1$, if $P$ necessarily requires flow values satisfying $\alpha_1$ to produce output.
• $\alpha_1$ and $\alpha_2$ are in disjunction w.r.t. $P$, denoted $\alpha_1 \circ \alpha_2$, if $P$ requires flow values satisfying either $\alpha_1$ or $\alpha_2$ (or both) to produce output. $\alpha_1$ is in unary disjunction w.r.t. $P$, denoted $\circ \alpha_1$, if $P$ possibly requires flow values satisfying $\alpha_1$ to produce output.

• $\alpha_1$ and $\alpha_2$ are in exclusive disjunction w.r.t. $P$, denoted $\alpha_1 \oplus \alpha_2$, if $P$ requires flow values satisfying either $\alpha_1$ or $\alpha_2$ but not both to produce output. $\alpha_1$ is in unary exclusive disjunction w.r.t. $P$, denoted $\oplus \alpha_1$, if $\alpha_1$ is a control flow intended to deactivate $P$.\(^4\) (We suggest extending the definition of unary exclusive disjunction to all information flows, both data and control flows, using semantics analogous to those given for output interfaces. We would say $\alpha_1$ is in exclusive disjunction w.r.t. $P$ if $P$ requires that flow values satisfying $\alpha_1$ do not exist in order for $P$ to execute. This preserves Kung's semantics for control flows and provides symmetry between input and output interface operator semantics.\)

We can also interpret an arbitrarily nested input interface as a logical expression defining when $P$ can execute, i.e., when the input interface is true, $P$ executes and produces output that satisfies $P$'s output interface. The label of an in-flow is true if there is a value on that flow.

Kung provides a graphical representation of information flow expressions with which to annotate DFD bubbles. In this thesis, we will simply list information flow expressions textually.

\(^4\)A control flow is defined similarly to Ward's signal, activation, and deactivation flows [59].
7.3.2 The Synchronization Problem

Figure 7.1 contains Kung's producer-consumer system composed of three DFD bubbles: a Producer, a Consumer, and an Update.BFC (buffer controller). The behavior of the Producer is given as follows:

The Producer is initially triggered by a signal on flow $s0$. When the Producer has generated a new item, it checks flow $bfc$ to see if the buffer is already full. If so, it sends a signal on flow $s2$ to inform Update.BFC the Producer is waiting for the flow $bfc$ to drop, otherwise it sends the new item on flow $x$ to the buffer, a signal on flow $s4$ to inform Update.BFC to update the buffer count, and a signal on flow $s3$ to trigger the production of a new item. If the Producer sent a signal on flow $s2$ to Update.BFC, then it waits until Update.BFC replies with a signal on flow $s1$. When the Producer receives a signal on flow $s1$, it sends the item, produced earlier, on flow $x$ to the buffer, a signal on flow $s4$ to Update.BFC, and a signal on flow $s3$ to itself. The Producer does not need to check the flow $bfc$ when it receives a signal on flow $s1$, because Update.BFC will only send a signal on flow $s1$ after it receives a signal on flow $s2$, indicating the buffer count has been decremented. When the Producer receives a signal on flow $s3$, it begins the whole process again.

A similar explanation can be given for the Consumer and Update.BFC bubbles.

The following scenario demonstrates the synchronization problem:

Suppose the Producer has received a signal on $s3$ and finds $bfc = max - 1$, where max is the limit on the size of the buffer. The Producer generates
an item and sends a signal on \( s3 \), a signal on \( s4 \), and the item on \( x \). At this point, both the Producer and Update_BFC can execute. Suppose the Producer receives the signal on \( s3 \) and accesses \( bfc \) before Update_BFC can increment \( bfc \). The Producer then generates and sends another item to the buffer, which is already full.

The essence of the synchronization problem is that when the Producer and Update_BFC can both execute, they are in a race to access \( bfc \). If the Producer wins and accesses the value of \( bfc \) before Update_BFC can increment \( bfc \), the buffer can overflow. In fact, if the Producer is fast enough, it could produce and add to the buffer several new items before Update_BFC increments the buffer count. An analogous underflow problem exists for the Consumer.
To repair this behavior, without drastically changing the semantics of the system, we need to have Update.BFC trigger the Producer and Consumer after updates to the buffer count are performed. This fix can be seen in Figure 7.2, where the flows s3 and s8 emanate from Update.BFC instead of allowing the Producer to trigger itself and the Consumer to proceed at random.

![Figure 7.2: Corrected producer-consumer Example](image)

This synchronization problem is an inherent problem in the producer-consumer system as characterized in this example. However, the synchronization of access to persistent and analog flow values is not an issue that needs to addressed at the specification level. We assume that "reads" and "writes" to persistent and analog flows happen instantaneously without overlap. Thus we assume process bubbles do not deadlock on access to persistent or analog flows. Sridhar and Hoare made similar assumptions [56]. Since consumable flows represent unbounded buffers and
there are no "critical section" concerns for analog and persistent flows, only inherent synchronization and deadlock problems need to be dealt with at the specification level. Thus deadlock and synchronization becomes not an issue of the particular implementation, but the logic of the specification.

7.3.3 Consistency Checking Deficiency

The deficiency identified in this Section pertains to both versions of the producer-consumer examples from the previous Section. We will work with Kung's original version [36]. Based on the description of the Producer's behavior, Kung provides the following information flow expressions:

input interface: \((s_0 \oplus s_3) \bullet (bfc \oplus s_1)\)

output interface: \(s_2 \oplus (x \bullet s_3 \bullet s_4)\)

The following four input patterns are the in-flow values which satisfy Kung's input interface.

1. \(s_0 \bullet bfc\)
2. \(s_3 \bullet bfc\)
3. \(s_0 \bullet s_1\)
4. \(s_3 \bullet s_1\)

The first pattern represents input which causes the Producer to generate the first item and send it to the buffer.\(^5\) The second pattern represents input which causes the

\(^5\) We could assume that when the Producer receives a signal on flow \(s_0\) that it is not necessary to test the flow \(bfc\) since the buffer is initially empty. However, the
Producer to generate subsequent items. This continues until the buffer becomes full and a signal is sent on flow \( s2 \). The remaining two patterns are infeasible. Consider pattern 3. If a signal is received on flow \( s0 \), the system is in some initial state, has not sent a signal on flow \( s2 \), and will not receive a signal on flow \( s1 \). Similarly for pattern 4, if a signal is received on flow \( s3 \), the Producer must have just previously sent a new item to the buffer and again will not receive a signal on flow \( s1 \). A signal on flow \( s1 \) is received only after the Producer has sent a signal on flow \( s2 \). In this case the Producer failed to send the item generated to the buffer and did not send a signal on flow \( s3 \). Thus Kung's input interface is incorrect.

We can characterize the dynamic behavior of the Producer by considering its state. Recall we have not incorporated the synchronization fix provided in Section 7.3.2. The Producer can only be in 1 of 3 states. These states are:

1. An initial state, where the Producer is waiting for a signal on flow \( s0 \), then produces an item and accesses the flow \( bfc \). If the value on flow \( bfc \) is less than some maximum, the Producer will send the new item on flow \( x \), send signals on flows \( s4 \) and \( s3 \), and will enter state 2. If the value on flow \( bfc \) is equal to the maximum, then the Producer will send a signal on flow \( s2 \) and enter state 3.

2. A success state, where the Producer has just previously sent an item on flow \( x \) and sent signals on flows \( s3 \) and \( s4 \). In this state, the Producer is waiting for a signal on flow \( s3 \), then produces an item and accesses the flow \( bfc \). If the value on flow \( bfc \) is less than some maximum, the Producer will send the new item on flow \( x \), send signals on flows \( s4 \) and \( s3 \), and will enter state 2. If the value

---

Producer's behavior was initially stated in this manner and the decomposition given later continues this characterization.
on flow $bfc$ is equal to the maximum, then the Producer will send a signal on flow $s2$ and enter state 3.

3. A blocked state, where the Producer has previously sent a signal on flow $s2$ and is waiting for a signal on flow $s1$. From this state, the Producer will only accept a signal on flow $s1$ and then send the generated item on flow $x$, send signals on flows $s4$ and $s3$, and enter state 2.

The following input interface correctly characterizes the Producer's behavior as described above.

input interface (corrected): $((s0 \oplus s3) \cdot bfc) \oplus s1$

---

Figure 7.3: A Decomposition of the Producer Process
In Figure 7.3 we show the DFD decomposition of the Producer bubble as given by Kung. The following list contains Kung's information flow expressions for the three bubbles in the DFD decomposition:

Generate_Item

input interface: \( s_0 \oplus s_3 \)

output interface: \( i_x \bullet s_x \)

Test_BFC

input interface: \( s_x \bullet \text{bfc} \)

output interface: \( s_y \oplus s_2 \)

Put_Buffer

input interface: \( i_x \bullet (s_y \oplus s_1) \)

output interface: \( x \bullet s_3 \bullet s_4 \)

Kung defines a mechanism for checking the consistency of DFD bubble decomposition by "balancing" the information flow expressions of the parent bubble with the information flow expressions of the child bubbles. Kung's consistency checking mechanism maps the parent's input interface to its output interface using the following 3 sets of rules:

1. logical rules: rules that are universally true and can be applied in any consistency proof.
2. *process rules:* rules that are true for the particular consistency proof at hand and are derived from the input interfaces and output interfaces of the child bubbles.

3. *inference rules:* rules that use logical rules and process rules to infer the intermediate and the final conclusions in a consistency proof. The inference rules provide implication, substitution, and derivation mechanisms for proofs using the logical and process rules.

To prove that the dynamic behavior of the network of child bubbles in Figure 7.3 is consistent with respect to the dynamic behavior of the Producer bubble in Figure 7.1, Kung uses the following rules:

**process rules:**

\[
\begin{align*}
\text{Generate.Item} & \vdash (s0 \oplus s3) \Rightarrow i_x \cdot s_x \\
\text{Test.BFC} & \vdash s_x \cdot \text{bfc} \Rightarrow s_y \oplus s_2 \\
\text{Put.Buffer} & \vdash i_x \cdot (s_y \oplus s_1) \Rightarrow x \cdot s_3 \cdot s_4
\end{align*}
\]

**logical rules:**

\[
\begin{align*}
\text{I1: } & \vdash \alpha \cdot \beta \Rightarrow \alpha \\
\text{I2: } & \vdash \alpha \cdot (\beta \oplus \delta) \Rightarrow (\alpha \cdot \beta) \oplus \delta \\
\text{I3: } & \vdash \alpha, \vdash \beta \Rightarrow \vdash \alpha \cdot \beta \\
\text{E1: } & \vdash \alpha \oplus \beta \Rightarrow \beta \oplus \alpha \\
\text{E2: } & \vdash \alpha \oplus (\beta \oplus \delta) \Rightarrow (\alpha \oplus \beta) \oplus \delta
\end{align*}
\]
Table 7.1: Kung's Consistency Proof

<table>
<thead>
<tr>
<th>step#</th>
<th>result inferred by the step</th>
<th>rules and steps involved</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>(\vdash (s0 \oplus s3) \bullet (bfc \oplus s1))</td>
<td>given input</td>
</tr>
<tr>
<td>2)</td>
<td>(\vdash i_x \bullet s_x \bullet (bfc \oplus s1))</td>
<td>Generate_Item, 1)</td>
</tr>
<tr>
<td>3)</td>
<td>(\vdash s_x \bullet (bfc \oplus s1))</td>
<td>I2, 2)</td>
</tr>
<tr>
<td>4)</td>
<td>(\vdash (s_x \bullet bfc) \oplus s1)</td>
<td>I2, 3)</td>
</tr>
<tr>
<td>5)</td>
<td>(\vdash (s_y \oplus s2) \oplus s1)</td>
<td>Test_BFC, 4)</td>
</tr>
<tr>
<td>6)</td>
<td>(\vdash s_y \oplus (s2 \oplus s1))</td>
<td>E2, 5)</td>
</tr>
<tr>
<td>7)</td>
<td>(\vdash s_y \oplus (s1 \oplus s2))</td>
<td>E1, 6)</td>
</tr>
<tr>
<td>8)</td>
<td>(\vdash (s_y \oplus s1) \oplus s2)</td>
<td>E2, 7)</td>
</tr>
<tr>
<td>9)</td>
<td>(\vdash i_x)</td>
<td>I2, 1)</td>
</tr>
<tr>
<td>10)</td>
<td>(\vdash i_x \bullet ((s_y \oplus s1) \oplus s2))</td>
<td>I3, 8) 9)</td>
</tr>
<tr>
<td>11)</td>
<td>(\vdash (i_x \bullet (s_y \oplus s1)) \oplus s2)</td>
<td>I2, 10)</td>
</tr>
<tr>
<td>12)</td>
<td>(\vdash (i_x \bullet s3 \bullet s4) \oplus s2)</td>
<td>Put_Buffer, 11)</td>
</tr>
</tbody>
</table>

Kung's consistency proof using the original incorrect input interface is shown in Table 7.1. However, when we replace the Producer's input interface with the corrected input interface the proof fails. Table 7.2 shows a proof attempt using the corrected input interface. To complete the proof we would need to change \((i_x \bullet s_y) \oplus s1\) into \((i_x \bullet (s_y \oplus s1))\) in order to apply the process rule for Put_Buffer. Note that this would be a reverse application of rule I2 and would not be sound. In fact from step (4), we can show that the proof fails, since \((i_x \bullet (s_y \oplus s2)) \oplus s1\) does not imply \((i_x \bullet (s_y \oplus s1)) \oplus s1\) \(\oplus s2\), consider \(i_x = s2 = false\) and \(s_y = s1 = true\). In the next Section we show that the problem with this proof is general and we suggest enhancements to correct the deficiency.

\(^6\)Kung calls these nonlogical rules.
Table 7.2: Consistency Proof Fails

<table>
<thead>
<tr>
<th>step#</th>
<th>result inferred by the step</th>
<th>rules and steps involved</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>((s0 \oplus s3) \bullet bfc \oplus s1)</td>
<td>given input</td>
</tr>
<tr>
<td>2)</td>
<td>((i2 \bullet s2) \bullet bfc \oplus s1)</td>
<td>Generate_Item, 1)</td>
</tr>
<tr>
<td>3)</td>
<td>((i2 \bullet (s2 \bullet bfc)) \oplus s1)</td>
<td>\bullet associativity, 2)</td>
</tr>
<tr>
<td>4)</td>
<td>((i2 \bullet (s2 \oplus s2)) \oplus s1)</td>
<td>Test_BFC, 3)</td>
</tr>
<tr>
<td>5)</td>
<td>((i2 \bullet s2) \oplus s1)</td>
<td>(E2, 5)</td>
</tr>
<tr>
<td>6)</td>
<td>((s2 \oplus (i2 \bullet s2)) \oplus s1)</td>
<td>(E2, 6)</td>
</tr>
</tbody>
</table>

7.3.4 Augmented Information Flow Expressions

We have shown that Kung's consistency checking mechanism does not work for the given producer-consumer example. The consistency checking mechanism fails for two reasons; (i) the use of implication rules like \(I2\) and (ii) there is no representation of persistent state in information flow expressions.

Kung states that consistency checking can be performed either forward (input to output) or backward (output to input). This would only be possible if we were restricted to equivalence rules and not implication rules like \(I2\). Note that rules \(E1\) and \(E2\) can be stated as equivalence rules. We can also view a process rule derived from the information flow expressions of a child process bubble as an equivalence rule, e.g., Generate_Item will produce output satisfying \(i2 \bullet s2\) if and only if it receives input satisfying \(s0 \oplus s3\).

Traditionally, SA bubbles were usually representations of procedural components of a sequential software system. The P-Specs required all their inputs and produced all their outputs. The value of the outputs was simply a function of the inputs. If we were restricted to procedural bubbles, equivalence rules might be sufficient for
consistency checking of dynamic behavior. However, the producer–consumer example uses the notion of process associated with distributed/concurrent systems. This notion of process is probably an essential perspective for writing specifications of distributed/concurrent systems. However, any consistency checking mechanism for the dynamic behavior of processes must include the notion of past behavior which we call persistent state. For example, if Generate_Item sends an item on flow $i_x$ and a signal on flow $s_x$, and then if Test_BFC receives the signal on flow $s_x$, finds the buffer full, and sends a signal on flow $s_2$, the item on the local flow $i_x$ still exists and needs to be accounted for when Put_Buffer later receives a signal on flow $s_1$. Accounting for the item on local flow $i_x$ is captured by state 3 mentioned in Section 7.3.3. In general, consistency checking between child and parent process bubbles requires modeling of persistent state for both the parent and child processes.

In the following discussion, we incorporate the synchronization fix provided in Figure 7.2 of Section 7.3.2. The corresponding decomposition is given in Figure 7.4.

We augment information flow expressions with an equivalence operator (denoted $i/o$). Using the equivalence operator, we bind an input interface to an output interface. We call each bound input/output interface pair an interface behavior. We construct an interface behavior expression for each state transition labeled with <from> and <to> states. For example, the Producer’s behavior defines the following five interface behaviors:

1. $<1> s_0 \bullet bfc \xrightarrow{i/o} x \bullet s_4 <2>$
2. $<1> s_0 \bullet bfc \xrightarrow{i/o} s_2 <3>$
Figure 7.4: A Corrected Decomposition of the Producer Process
3. \(<2\) \(s3 \bullet bfc \xrightarrow{i/o} x \bullet s4 <2\>

4. \(<2\) \(s3 \bullet bfc \xrightarrow{i/o} s2 <3\>

5. \(<3\) \(s1 \xrightarrow{i/o} x \bullet s4 <2\>

Process rules are derived from the interface behaviors of each child bubble. For example, the interface behavior for Generate_Item is \(s0 \oplus s3 \xrightarrow{i/o} i_x \bullet s_x\). We drop the \(<\text{from}\>\) and \(<\text{to}\>\) state labels from interface behaviors of bubbles with only one state. We call these bubbles simple processes. We replace expressions containing the disjunction operator with an equivalent expression using only conjunction and exclusive disjunction, i.e., \((x \circ y) \equiv (x \oplus y \oplus (x \bullet y))\). We then distribute exclusive disjunction over equivalence to derive the following process rules for Generate_Item:

**Generate_Item**

\[\begin{align*}
\text{G11: } & s0 \xleftrightarrow{i/o} i_x \bullet s_x \\
\text{G12: } & s3 \xleftrightarrow{i/o} i_x \bullet s_x
\end{align*}\]

We derive the following process rules in a similar manner:

**Test.BFC**

\[\begin{align*}
\text{TB1: } & s_x \bullet bfc \xleftrightarrow{i/o} s_y \\
\text{TB2: } & s_x \bullet bfc \xleftrightarrow{i/o} s2
\end{align*}\]

**Put Buffer**

\[\begin{align*}
\text{PB1: } & i_x \bullet s_y \xleftrightarrow{i/o} x \bullet s4
\end{align*}\]

\(^7\) Note that the state transition model of each child bubble contains a single state.
A process rule is just a special case of an interface behavior. The replacement of disjunction and distribution of exclusive disjunction results in process rules that require all their inputs and produce all their outputs. Thus a process rule behaves procedurally, i.e., like a P-Spec in traditional SA. Conversely, a P-Spec in traditional SA can be modeled with a single state with one transition.

Process rules define the dynamic behavior of each child bubble. To prove the consistency of the dynamic behavior of the parent bubble with respect to the dynamic behavior of the network of child bubbles, we need to prove that the network of child bubbles can exhibit each of the interface behaviors of the parent bubble starting from the initial state. Recall that the dynamic behavior of the parent bubble is given in terms of a finite number of states. A parent interface behavior \( \langle i \rangle \ X \leftrightarrow Y \) \( \langle j \rangle \) is consistent if using logical equivalence and process rules we can transform \( (X \leftrightarrow Y) \) into \( (X' \leftrightarrow \text{true}) \) where \( X' \) does not contain the label of any flows into or out of the parent bubble. Thus we can generate all out-flows appearing in \( Y \) and we required all in-flows appearing in \( X \). \( X' \) may be empty or contain labels of flows appearing only in the DFD decomposition, i.e., local flows. The proof process continues by considering each of the behaviors from state \( j \). In the case that \( X' \) is non-trivial, i.e., \( X' \neq \text{true} \), then \( X' \) must be conjuncted with the input interface of every interface behavior that starts from state \( j \). For example, suppose we have an initial parent behavior \( \langle 1 \rangle \ \text{in-flow}_1 \leftrightarrow \text{out-flow}_1 \langle i \rangle \). Further suppose it can be shown that in-flow \( i \) = child-flow \( 1 \) \( \bullet \) out-flow \( 1 \langle i \rangle \). Then substituting for in-flow \( 1 \), we have \( \langle 1 \rangle \ \text{child-flow}_1 \leftrightarrow \text{out-flow}_1 \langle i \rangle \) which is equivalent to \( \langle 1 \rangle \)
child-flow_j \overset{i/o}{\leftrightarrow} true \overset{i}{\bullet} \text{in-flow}_j \overset{i/o}{\leftrightarrow} \text{out-flow}_j \overset{k}{\bullet} \text{we need to prove \((<i> \text{child-flow}_1 \bullet \text{in-flow}_j \overset{i/o}{\leftrightarrow} \text{out-flow}_j \overset{k}{\bullet} \).}

These non-trivial \((X')\)'s represent local memory of past behavior, i.e., persistent state. In cases where a child bubble is not a simple process, i.e., has more than one state, then its current state defines which of its process rules are currently applicable in the proof and is also part of the representation of persistent state. We call the conjunction of a non-trivial past behavior with the next state's input interface an application of the persistent state rule (denoted \(PS \)). We refer to the application of any standard logical rule (e.g., associativity of \(\bullet\), etc.) as an application of the logical equivalence rule (denoted \(LE \)). In Table 7.3, we demonstrate the consistency proof of the Producer decomposition. Since all child bubbles are simple processes, every process rule is always applicable and we can ignore child bubble states. Note that interface behaviors for transitions from state 1 and 2 to state 3 result in non-trivial persistent state.

Consider Test_BFC and states 1 and 2 of the Producer. There appears to be possible nondeterministic behavior. When proving consistency of the interface behaviors for states 1 and 2, we made an apparently nondeterministic choice to apply either rule TB1 or TB2. Making the opposite choice would not have worked and we would have to backtrack to apply the other process rule. However, if we associate each of these interface behaviors with a predicate on \(bfc\), we see that the choice is no longer nondeterministic and was made correctly.

We have demonstrated that proofs of consistency between a network of child bubbles and a corresponding parent bubble are feasible if we treat input interface expressions as equivalent to output interface expressions and we model persistent
Table 7.3: Producer Consistency Proof

<table>
<thead>
<tr>
<th>step#</th>
<th>result inferred by the step</th>
<th>rules and steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>proof: from state 1 (trivial persistent state)</td>
<td>initial state</td>
</tr>
<tr>
<td></td>
<td><strong>result inferred by the step</strong></td>
<td><strong>rules and steps</strong></td>
</tr>
<tr>
<td></td>
<td>proof: from state 1 (trivial persistent state)</td>
<td>initial state</td>
</tr>
<tr>
<td>(1.1.1)</td>
<td>$&lt;1 &gt; s0 \bullet bfc \xrightarrow{i/o} x \bullet s4 &lt;2 &gt;$</td>
<td>given behavior 1, (1)</td>
</tr>
<tr>
<td>(1.1.2)</td>
<td>$&lt;1 &gt; i_x \bullet s_x \bullet bfc \xrightarrow{i/o} x \bullet s4 &lt;2 &gt;$</td>
<td>GI1, (1.1.1)</td>
</tr>
<tr>
<td>(1.1.3)</td>
<td>$&lt;1 &gt; i_x \bullet s_y \xleftarrow{i/o} x \bullet s4 &lt;2 &gt;$</td>
<td>TB1, (1.1.2)</td>
</tr>
<tr>
<td>(1.1.4)</td>
<td>$&lt;1 &gt; x \bullet s4 \xrightarrow{i/o} x \bullet s4 &lt;2 &gt;$</td>
<td>PB1, (1.1.3)</td>
</tr>
<tr>
<td>(1.1.5)</td>
<td>$&lt;1 &gt; true \xleftarrow{i/o} true &lt;2 &gt;$</td>
<td>$\mathcal{E}$, (1.1.4), goto (2)</td>
</tr>
<tr>
<td>(1.2.1)</td>
<td>$&lt;1 &gt; s0 \bullet bfc \xrightarrow{i/o} s2 &lt;3 &gt;$</td>
<td>given behavior 2, (1)</td>
</tr>
<tr>
<td>(1.2.2)</td>
<td>$&lt;1 &gt; i_x \bullet s_x \bullet bfc \xrightarrow{i/o} s2 &lt;3 &gt;$</td>
<td>GI1, (1.2.1)</td>
</tr>
<tr>
<td>(1.2.3)</td>
<td>$&lt;1 &gt; i_x \bullet s2 \xrightarrow{i/o} s2 &lt;3 &gt;$</td>
<td>TB2, (1.2.2)</td>
</tr>
<tr>
<td>(1.2.4)</td>
<td>$&lt;1 &gt; i_x \xrightarrow{i/o} true &lt;3 &gt;$</td>
<td>$\mathcal{E}$, (1.2.3), goto (3)</td>
</tr>
<tr>
<td>(2)</td>
<td>proof: from state 2 (trivial persistent state)</td>
<td>(1.1.5), (2.3.5), (3.5.3)</td>
</tr>
<tr>
<td>(2.3.1)</td>
<td>$&lt;2 &gt; s3 \bullet bfc \xrightarrow{i/o} x \bullet s4 &lt;2 &gt;$</td>
<td>given behavior 3, (2)</td>
</tr>
<tr>
<td>(2.3.2)</td>
<td>$&lt;2 &gt; i_x \bullet s_x \bullet bfc \xrightarrow{i/o} x \bullet s4 &lt;2 &gt;$</td>
<td>GI2, (2.3.1)</td>
</tr>
<tr>
<td>(2.3.3)</td>
<td>$&lt;2 &gt; i_x \bullet s_y \xleftarrow{i/o} x \bullet s4 &lt;2 &gt;$</td>
<td>TB1, (2.3.2)</td>
</tr>
<tr>
<td>(2.3.4)</td>
<td>$&lt;2 &gt; x \bullet s4 \xrightarrow{i/o} x \bullet s4 &lt;2 &gt;$</td>
<td>PB1, (2.3.3)</td>
</tr>
<tr>
<td>(2.3.5)</td>
<td>$&lt;2 &gt; true \xrightarrow{i/o} true &lt;2 &gt;$</td>
<td>$\mathcal{E}$, (2.3.4), goto (2)</td>
</tr>
<tr>
<td>(2.4.1)</td>
<td>$&lt;2 &gt; s3 \bullet bfc \xrightarrow{i/o} s2 &lt;3 &gt;$</td>
<td>given behavior 4, (2)</td>
</tr>
<tr>
<td>(2.4.2)</td>
<td>$&lt;2 &gt; i_x \bullet s_x \bullet bfc \xrightarrow{i/o} s2 &lt;3 &gt;$</td>
<td>GI2, (2.4.1)</td>
</tr>
<tr>
<td>(2.4.3)</td>
<td>$&lt;2 &gt; i_x \bullet s2 \xrightarrow{i/o} s2 &lt;3 &gt;$</td>
<td>TB2, (2.4.2)</td>
</tr>
<tr>
<td>(2.4.4)</td>
<td>$&lt;2 &gt; i_x \xrightarrow{i/o} true &lt;3 &gt;$</td>
<td>$\mathcal{E}$, (2.4.3), goto (3)</td>
</tr>
<tr>
<td>(3)</td>
<td>proof: from state 3 (persistent state $i_x$)</td>
<td>(1.2.4), (2.4.4)</td>
</tr>
<tr>
<td>(3.5.1)</td>
<td>$&lt;3 &gt; i_x \bullet s_1 \xrightarrow{i/o} x \bullet s4 &lt;2 &gt;$</td>
<td>given behavior 5, $\mathcal{P}S$, (3)</td>
</tr>
<tr>
<td>(3.5.2)</td>
<td>$&lt;3 &gt; x \bullet s4 \xrightarrow{i/o} x \bullet s4 &lt;2 &gt;$</td>
<td>PB2, (3.5.1)</td>
</tr>
<tr>
<td>(3.5.3)</td>
<td>$&lt;3 &gt; true \xrightarrow{i/o} true &lt;2 &gt;$</td>
<td>$\mathcal{E}$, (3.5.2), goto (2)</td>
</tr>
</tbody>
</table>
state of all bubbles. The proof mechanism given above does not guarantee consistent
dynamic behavior, but does prove that such consistent behavior is possible within
the scope of the semantics provided by these augmented information flow expressions.
The proof mechanism described in the previous section can be reduced to an graph
edge covering algorithm. In the worst case, each transition (edge) must be proved with
every possible representation of persistent state. Persistent state is stated in terms
of information flow expressions over flow labels appearing only in the decomposition
(child-flows) and not appearing as in-flows or out-flows of the parent bubble and
the current state of each child bubble. The number of child-flows is of course finite
and thus there is only a finite number of possible truth assignments to the child-
flow labels. Likewise each child bubble has a finite number of possible states. This
bounds the number of edge traversals required for each transition. Using a back
tracking algorithm to handle non-deterministic rule applications would allow us to
automate the proof mechanism.

Observe that the proof mechanism requires a finite state representation of the
parent bubble. We can argue that this representation obviates the need to decompose.
However, it may be possible to turn the proof mechanism "upside down" and generate
the dynamic behavior of a parent bubble from the dynamic behavior of the children.\(^8\)
In this way, we need only state the dynamic behavior of P-Spec bubbles. The difficulty
of stating a particular bubble's dynamic behavior may be indicative of the need to
further decompose that bubble into a DFD.

The input interface defined by Kung served as a guard on the execution of a
bubble. This characterization is continued in the development of interface behaviors

\(^8\)Kung made a similar observation.
and process rules. However, there may be additional functional requirements which guard the execution of a bubble. For example, a bubble B with in-flows x, y, and z and input interface \(((x \cdot y) \circ z)\) may be executable only when either the flow value on x is greater than or equal to the flow value on y and z is false (no flow value exists) or when x, y, and z are all true.\(^9\) We can include predicates on flow values within information flow expressions to capture these functional guards.

The other component of bubble behavior, the functional component, relates the value of in-flows to the value of out-flows. We could further augment interface behaviors with first order logic assertions which define the value of out-flows produced in terms of the values of in-flows required for a particular transition. However, including predicate guards or first order logic assertions in the proof mechanism would make the mechanism second order and non-automatable. In Section 7.4 we extend interface behaviors with first order logic assertions to specify the functional component of bubble behavior.

### 7.4 Modified Stated Transition Diagram Process Specifications

In Section 7.3.4 we replaced the use of disjunction with equivalent exclusive-disjunction of conjunctions and distributed exclusive-disjunction over the interface equivalence operator. The result was that each process rule defined a procedure— it required all its inputs and produced all its outputs. To specify the functional behavior of a procedure in SPECS, preconditions and postconditions written in first order logic are provided. However, the semantics of the precondition in SPECS is the former of Yourdon's two alternatives—the user may assume the precondition is true when the

\(^9\)We assume x and y carry numeric flow values.
procedure is executed.

In a process rule the input interface side (which we call the enabling condition) of the expression has the semantics of Yourdon's second alternative—when the expression is true the procedure executes. When we combine this execution enabling semantics with first order logic, in-flow labels take on a dual semantics. Each label appearing in an enabling condition means that the value is required for application of this process rule and the label represents that value in the evaluation of the enabling condition's truthness. In the case of an analog or persistent flow, the label represents a read reference of the flow value. In the case of a consumable flow, the label represents the reception of a flow value. If the entire enabling condition is true, the consumable flow value is removed from the flow. Since we have distributed Kung's exclusive-disjunction over the interface equivalence operator, we avoid mixing the semantics of Kung's conjunction, disjunction, and exclusive-disjunction with the similar operators in first order logic. However, there may be in-flow values which are not restricted by logical conditions in the enabling condition, but are required for computation of the resulting out-flow values. Since in-flow labels represent the actual flow value and not the existence of a flow value, any in-flow labels which appear in the output interface part of a process rule (which we call the postcondition) also represent required flow values.

Flows that represent control-flow-like signals can be modeled by a 0-tuple. To assert their presence we can use an equality test of the flow label with the special constant Null. However, we may want to assert its absence, similar to Kung's unary semantics for \( \ominus \). To do this we provide a special consumable flow operator \( \exists \) and interpret \( \exists(X) \) as true if and only-if a flow value exists on the consumable
flow $X$. To assert their absence, we negate the $\exists$ operator, i.e., $\neg \exists X$. The operator $\exists$ can actually be applied to any consumable flow, not just consumable flows whose flow value domain is $\{ \text{Null} \}$.

Both the dynamic and functional specification of a DFD bubble can be provided by a state transition diagram whose edges are predicated by process rules. However, if a DFD bubble decomposes into another DFD then either the state diagram or the DFD diagram is redundant. They should both specify the same behavior. Thus for specification purposes, we limit the use of predicated state transition diagrams to P-Specs, i.e., a P-Spec is specified by a predicated state transition diagram. Furthermore, the set of flow labels appearing in any process rules predicking any edge must be equal to the union of the in-flow and out-flow labels. The state transition diagram specification can be given textually or diagrammatically. In the case, where a P-Spec has only one state, we suggest using a textual representation.

The particular first order logical assertions are limited to the use of operations provided by the ADT specification of flow labels. Modularity of first order logical assertions can be provided by expression definitions local to the P-Spec. We incorporate these modification into the formal model of SA by modifying the Type Definition 11 as provided in Type Definition 23.

**Type Definition 23**

$$\text{P-Spec} \quad = \quad 4\text{-tuple}(\text{ProcessLabel} : \text{Label},$$
$$\text{InFlows} : \text{set of FlowType},$$
$$\text{OutFlows} : \text{set of FlowType},$$
$$\text{Behavior} : \text{PredicatedSTD});$$

A predicated state transition diagram is defined by Type Definitions 24 and 25. The BNF descriptions of an expression definition and a first order boolean expression
are contained within the BNF of a SPECS ADT are not given.

**Type Definition 24**

PredicatedSTD = 4-tuple(

States: set of Label,
InitialState: Label,
Transitions: set of P-Rule,
ExprDefs: set of BNF[ExpressionDefinition]);

**Type Definition 25**

P-Rule = 4-tuple(

From: Label,
To: Label,
Enable: BNF[BooleanExpression],
Post: BNF[BooleanExpression]);

Expression Definition 10 provides the set of flow labels appearing in a boolean expression. Expression Definition 11 asserts the existence of a path of transitions from one state to another in a set of P-Rules.

**Expression Definition 10**

define ExpFlowLabels(E: BNF[BooleanExpression]) as set of ModifiedLabel such that

ExpFlowLabels = \{ x | x was found in a parse of E\}

Invariant Property 16 provides the usual restrictions on di-graphs such as STDs and assure that flow labels appearing in P-Rules match with the in-flows and out-flow of the P-Spec.
Expression Definition 11

define \( IsAPath(\text{From}: \text{Label}, \text{To}: \text{Label}, Edges: \text{set of P-Rule}) \) as boolean

\[ IsAPath \equiv \exists p : \text{sequence of P-Rule}[\]
\[ \forall i[1 \leq i \leq \text{length}(p) \Rightarrow p_i \in Edges] \]
\[ \land p_1 = \text{From} \land p_{\text{length}(p)} = \text{To} \]
\[ \land \forall j[1 \leq j < \text{length}(p) \Rightarrow \text{To}(p_j) = \text{From}(p_{j+1})] \]

Invariant Property 16

(for any \( P \) of type P-Spec):
\( \text{InitialState(\text{Behavior}(P))} \in \text{States(\text{Behavior}(P))} \)
\[ \land \left( \bigcup_{t \in \text{Transitions(\text{Behavior}(P))}} (\text{From}(t) \cup \text{To}(t)) \right) = \text{States(\text{Behavior}(P))} \]
\[ \land \forall s[s \in \text{States(\text{Behavior}(P))} \Rightarrow \]
\[ \text{IsAPath(\text{InitialState(\text{Behavior}(P))}, s, \text{Transitions(\text{Behavior}(P))})} \]
\[ \land \left( \bigcup_{t \in \text{Transitions(\text{Behavior}(P))}} \text{ExpFlowLabels(Enable}(t)) \right) \subseteq \text{InFlows}(P) \]
\[ \land \left( \bigcup_{t \in \text{Transitions(\text{Behavior}(P))}} (\text{ExpFlowLabels(Enable}(t)) \cup \text{ExpFlowLabels(Post}(t))) \right) \]
\[ = (\text{InFlows}(P) \cup \text{OutFlows}(P)) \]
CHAPTER 8. FORMALIZING STORES AND TERMINATORS

8.1 Overview of Other DFD Components

Stores and terminators are the two remaining components of DFDs that we have not discussed. In this chapter we present some of the issues we face when trying to interpret the behavior of stores and terminators. We will also suggest some rudimentary formalizations of stores and terminators and identify issues which require further research. It is not our intention to develop a formalized interpretation of stores and terminators to the same extend devoted to flows and bubbles.

8.2 Stores

In traditional SA, stores are specified by an associated DDE. A composition expression was used to describe the structure of the data associated with a store. Stores are considered passive objects, they do not initiate their out-flows, nor request their in-flows. The in-flows and out-flows of a store must also "balance" with respect to the DDE definition of the store. An in-flow or out-flow could be labeled by the same label as the store. For example, store bfc of the producer-consumer system from Chapter 7 had three out-flows labeled bfc and a single in-flow labeled bfc'. The prime notation (i.e. bfc') was Kung's notation for representing the placement of a new value in the store bfc [36]. An in-flow or out-flow could also be labeled by a
subordinate of a store's composition expression definition. For example, store buffer
of the producer-consumer system could be defined in the DDE as $0\{x\}max$.

### 8.2.1 Do We Need Stores?

Stores are said to represent data at rest. In some way, data associated with a
store is thought to be less active than data associated with a flow. We feel this is
incorrect. Data associated with a store serves the same role as data associated with a
persistent flow. Stores simply serve to provide a more expressive notation for objects
that persist over time and are modified and accessed by several different bubbles.

Recall the corrected producer-consumer system from Chapter 7. Two stores
are present in the first DFD. Consider first the store labeled buffer. The buffer
store represents a FIFO queue of objects having the same structure as the objects
associated with the flow labeled $x$. The description of the Producer bubble suggests
that the flow labeled $x$ is a consumable flow. In this case the store buffer is redundant,
as the $x$ flow itself already provides the FIFO ordering of $x$ structured objects.

The second store, labeled bfc, represents the persistent count of the number
of objects in the buffer. All three bubbles have access to this value, but only Update_BFC can modify it. This store can be modeled by a diverging persistent flow
written by Update_BFC and read by all three bubbles. Based on these two obser-
vation, the (corrected) producer-consumer DFD can be specified without the use of
stores as shown in Figure 8.1. However, in the case where a large number of bubbles
can modify a particular value and a large number of bubbles can access a particular
value, using a store, instead of converging and diverging persistent flows, may help

---

1 The context diagram for the producer-consumer system was not given.
unclutter the DFD thus improving readability.

![Figure 8.1: A Storeless Producer-Consumer DFD](image)

8.2.2 Balancing Flows

Since stores are similar to flows, we might want to specify stores using a SPECS ADT. However, what do in-flows and out-flows then represent? In the case of a flow having the same label as the store, the interpretation could represent a reference or replacement of the entire store's value, but by which ADT operation? Flows labeled by other labels might represent formal parameter types for the operations a bubble might perform on a store. The problem then becomes which operation does each flow or set of flows intend as the recipient?

There is certainly the potential for several ADT operations having the same input or output parameters. Labeling flows with operation names is not a valid alternative because it changes the overall semantics of a flow. A flow should represent data
communication, not flow of control to a subroutine as in a calling chart. However, operation names can be used in the enabling-conditions and postconditions associated with P-Rules. The balancing of flows between a store and a bubble would then need to check that all and only parameter types and result types of store operations appearing in that bubble's P-Rules, label flows between that bubble and the store. These flows should have the appropriate direction—store in-flow for parameter and store out-flow for result. Note that a flow labeled the same as the store represents the use of an operation which inputs or outputs an object of the same structure as the store.

8.2.3 Concurrent Stores

Although we have not provided explicit parallelism, there does exist the potential for defining concurrent execution of P-Specs in SA specifications. Reads and writes to persistent and analog flows do not provide a locking mechanism to prevent access to a flow by another P-Spec.\(^2\) This type of behavior can be critical for providing database consistency or avoiding deadlock. As persistent and analog flows do not provide this ability, we might consider providing this ability exclusively to stores. This allows the continued traditional use of flows (albeit our formalized interpretation) and provides for distributed concurrency control with in the scope of SA specifications.

We suggest that SPECS ADTs associated with stores also include some statement of which operations can be performed in parallel or other constraints such as sequencing of operations. Two declarative style methods for this type of specification are open path expressions and predicate path expressions [46]. Both types of

\(^2\)For consumable flows the issue is mute, since individual consumable flow values are never accessed by more than one P-Spec.
path expressions provide declarative synchronization specifications for controlling the execution of operations on an ADT by P-Specs.

Another possible concurrency representation is provided directly by a DFD. In this case, a store is decomposed into a DFD as in Figure 8.2. Each operation on the store is provided by a passive bubble. The passive bubbles never initiate communication with off-page entities and thus the store maintains its passive role. Constructor bubbles appear as writers to a persistent flow having the same label (i.e. structure) as the original store. Constructor bubbles only have off-page in-flows which represent parameters for modifying the contents of the store and have access to the value of the store. Selector bubbles appear as readers of this persistent flow. Selector bubbles have both off-page in-flows representing information for query formation and off-page out-flows representing the query results. Thus a store label is associated with both a DDE for the ADT representation of the persistent flow and a DFD to represent the concurrent access to the store. Synchronization communication can be provided by flows between the bubbles. To minimize hierarchical complexity, we assume the bubbles are all specified by P-Specs. Flow label modifiers can be used to distinguish between bubbles having the same set of in-flows or out-flows.

We might also consider the passive bubbles as defining a bubble class. That is we might allow for multiple threads of execution in a particular bubble, each with its own persistent state. Each activation of a thread of execution in a passive bubble would place the P-Spec in an initial state. From there the passive bubble would complete some modification and/or selection operation on the store, report the results to the off-page activating bubble, and terminate the thread of execution. The passive bubbles may be responsible for maintaining consistency in the store as viewed by the
external (activating) bubbles. The activating bubble could either suspend and wait on the passive bubble—like an imperative procedure call—or fork the passive bubble and continue with its own computations.

8.2.4 The Future of Stores

Stores, and flows between stores and bubbles, are clearly different than flows between bubbles. Based on the discussion of issues presented in this thesis, we identify 7 issues to be resolved by future research:

- How does a store represent data?
- How is this representation fundamentally different from flows? (Is it?)
- How are store operations represented and executed?
• What is the interpretation of flow balancing with respect to stores?

• If stores provide concurrency control, are the operations themselves processes (forked) or procedures (suspend & wait)?

• Are multiple threads of execution allowed for store bubbles?

• How is concurrency controlled?

In Chapter 9, we ignore the role of the store in the operational semantics of a DFD.

### 8.3 Terminators

Terminators are easier to resolve than stores. It is intended in an SA model that less is known about the potential behavior of terminators. Terminators are outside the system. However, the flows between terminators and the context DFD bubble represent the interface between the system and users, devices, and other systems that represent the external world. To develop a system that will respond correctly to the in-flows from the terminators, we should at least have some idea of the temporal ordering of these communications.

Terminators behave like bubbles in that they may actively initiate communication and they may have persistent state. We generally cannot specify the functionality of their communications, however, we may be able to predict which type of communications (i.e. flows values) will appear next. For example, after receiving a value on the \textit{customer-name} in-flow, we expect (because we have prompted for it) the next flow value to appear will be on the \textit{customer-phone} in-flow. This type of temporal dynamic ordering can be described by the augmented information flow expressions and state transition diagrams presented in Chapter 7.
Our suggestion for the specification of a terminator is provided by Type Definition 26. In the case that a terminator is another system, e.g., a file system accessed by a spelling checker system, a more detailed representation can be provided by the other system's specification.

**Type Definition 26**

Terminator = 2-tuple( TermName:Label, 
Behavior:DecoratedSTD);

Type Definition 27 defines a STD similar to the predicated STD defined for a P-Spec, except local expression definitions are not allowed and the enabling and post conditions are limited to the interface behaviors presented in Chapter 7. Type Definition 28 provides transitions decorated with interface behavior expressions. Recall an interface behavior expression is a grouping of Kung’s input interface and output interface expressions. Table 8.1 provides a BNF description of an interface behavior expression.

**Type Definition 27**

DecoratedSTD = 3-tuple( States:set of Label, 
Initial:Label, 
Transitions:set of InterfaceRule);

**Type Definition 28**

InterfaceRule = 3-tuple( From:Label, 
To:Label, 
Interface:BNF[InterfaceBehavior]);
Table 8.1: Interface Behavior Syntax

<table>
<thead>
<tr>
<th>InterfaceBehavior</th>
<th>::=</th>
<th>InterfaceExpr $\leftarrow\rightarrow$ InterfaceExpr</th>
</tr>
</thead>
<tbody>
<tr>
<td>InterfaceExpr</td>
<td>::=</td>
<td>[ InterfaceExpr ] operator InterfaceExpr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>modifier</td>
<td>::=</td>
<td>'&lt;'identifier'&gt;,'</td>
</tr>
<tr>
<td>operator</td>
<td>::=</td>
<td>\bullet</td>
</tr>
</tbody>
</table>
CHAPTER 9. SEMANTICS OF SA SPECIFICATIONS

9.1 Overview

In previous chapters we have presented interpretations of the behavior of individual components of an SA specification. In this chapter we present a operational semantics for animating an SA specification. As mentioned in Chapter 8 we exclude the use of stores from our operational semantics.

The animation of a SA specification is similar to the execution of a data flow program [19]. Process bubbles in SA DFDs are analogous to actors in a data flow program. The primary difference is in the complexity of the functional behavior of the process bubbles. Actors provide primitive computational abilities similar to operators and control statements in an imperative program. Theoretically we could specify a system by decomposing the DFD hierarchy to the point where the P-Specs could be specified by data flow program actors. However, this defeats the purpose of a specification. We don’t want to state how to solve the problem, but rather state the behavior of the solution. This does not prevent the implementation of a SA specification from being targeted for a data flow machine [50].
9.2 An Operational Interpretation

The operational semantics of an SA specification are based on the “football field sized” DFD represented by the hierarchy of DFD and P-Spec processes. Recall that in the “football field sized” DFD, each bubble represents a P-Spec. We consider each P-Spec as an abstraction on a process in a distributed or concurrent system. Although we want to stress that a P-Spec is not an implementation process, we could abstractly think of each P-Spec as being assigned a physical processor. As part of this abstraction we would assume that no process (or processor) fails in anyway. Thus our operational semantics do not include, nor depend on, a particular processor control or recovery control mechanism. Our operational semantics are based solely on the functionality and dynamic timing provided by the specification.

The operational semantics for an SA specification are similar to the execution of a Petri net [51]. Thus formalized SA specifications are well-suited for the specification of distributed systems [63]. The primary difference between a Petri net execution and our operational semantics is the association of value with tokens and the use of first order logical assertions on these values for transition firing rules.

In the “football field sized” DFD flows represent Petri net places and flow values represent tokens. An analog or persistent flow place will always contain a single token. Tokens associated with consumable flow places are ordered by arrival time.

Each P-Spec represents a Petri net transition, modified to include first order predicate logic in its firing rule, and a single place serving both as an input place and an output place for the transition. The single place will contain a single token whose value is the state of the P-Spec. The transition’s firing rule is based on the presence and value of tokens in the places represented by in-flow and the value of the
token in the place which represents state. Each terminator also represents a Petri
net transition. However, the firing rules for terminators are based solely on token
presence and state and are not predicated by token values.

The initial marking is defined by placing a state token containing the initial state
in each state place and placing a single token having the value \texttt{undefined} in each
analog and persistent flow place. Consumable flow places are initially empty.

A single entity is chosen from the set of all terminators and P-Specs that are
able to fire. A P-Spec is able to fire if its in-flow places contain tokens whose values
allow the enabling condition of any P-Rule from the current state to evaluate true.
A terminator is able to fire if its in-flow places contain tokens that allow the input
side of an interface behavior of any InterfaceRule from the current state to evaluate
true.

The entity chosen is then fired. When a P-Spec fires it consumes those con-
sumable in-flow tokens that resulted in its enablement. The P-Spec then produces
consumable out-flow place tokens and replaces analog and persistent out-flow place
tokens with tokens. The resulting tokens have values which satisfy the postcondition
of the P-Rule which enabled that P-Spec. When a terminator fires it consumes those
consumable in-flow tokens that resulted in its enablement. The terminator then pro-
duces consumable out-flow place tokens and modifies analog and persistent out-flow
place tokens based on the output side of the interface expression of the InterfaceRule
which enabled that terminator. The value of tokens produced by the firing of a
terminator are not specified by the SA specification. In such cases we assume the
animator (user) of the SA specification will provide the appropriate value. The firing
of a P-Spec or Terminator happens instantaneously.
The resulting marking defines a new system state. The selection and firing of an enabled P-Spec or terminator is repeated indefinitely as long as there exists an enabled P-Spec or terminator to fire.
CHAPTER 10. CONCLUSIONS

10.1 Summary

In this thesis:

1. We formally defined the structure of a traditional SA specification and developed and defined the structure of a formalized SA specification. The definition of the structure of a formalized SA specification is based on modifications of traditional SA specifications to include our enhancements, improvements, and interpretation of traditional SA specifications. The definition of the structure of SA specifications is given in the formal specification language SPECS.

2. We developed an interpretation of flow labels that minimizes redundancy in the DDE and allows for the synthesis of object-oriented specifications with SA specifications. This is because we interpret flow labels in the DD as types, in particular SPECS ADTs.

3. We developed an interpretation of three flow value behaviors—analogue, persistent, and consumable. This allows us to represent flow values similar to the placement of tokens in Petri net places [51].

4. We developed an interpretation of flow abstraction that allows for balancing of flow abstraction and provides an interpretation of flow value timing. Flow
value timing also contributes to the Petri net like representation of flow values.

5. We formalized the DDE specification of flow and store value type structure by incorporating SPECS ADTs. This formalization is based on well-defined mathematical objects rather than the traditional lexical description of data values. This formalization provides a synthesis of object-oriented specifications with SA specifications.

6. We formalized the specification of P-Spec functionality and timing using the finite state Mealy machine and first order predicate calculus assertions. This formalization allows for an assertional-based characterization of process bubbles as true distributed/concurrent processes. By characterizing process bubbles as true distributed/concurrent processes, we obviate the need for real-time extensions to SA specifications. This is accomplished by removing the distinction between control values and data values, and by applying the finite state machine specification of control bubbles directly to the specification of P-Specs. The first order assertions provide a mathematically precise method for specifying P-Spec functionality. The state machine representation and enabling semantics of input assertions provides a Petri net like execution of DFD bubbles.

7. We discussed the potential for formalized specification of store behavior. In particular, we identified that a flow between a process bubble and a store may require a different interpretation than a flow between two process bubbles. We also identified the potential for additional concurrency in the modification of and reference to a store's value.
8. We provided a formalized specification of terminators that takes into account the behavior and timing of terminators with respect to flow value production and consumption.

9. Based on the formalization of informally defined SA components, the formal interpretation of SA components' behavior, and the formal definition of the structure of a formalized SA specification, we suggested an operational semantics for an SA specification as a whole. These operational semantics provide a token based animation of an SA specification analogous to the token based execution of a Petri net. However, the assertion based state machine specification of process bubble timing and functionality combined with the timing of flow value abstraction provides a higher order abstraction of system functionality than can be expressed by a traditional Petri net. In particular, in a traditional Petri net, tokens do not have values and transition executions are not based on satisfaction of first order assertions.

10.2 Future Work

The first task is to complete the formalization of SA specifications by providing a rigorous characterization of stores and their behavior that lends itself to the animation of SA specifications. One other potential enhancement of formalized SA specifications is the representation of sub-system classes. We may want certain groups of DFD components to represent multiple occurrences of units of the system that have the same behavior. For example, in a library circulation system a certain group of process bubbles, stores, and terminators might represent a branch library's part of the system. In the model we may want to reflect that this branch sub-system oc-
curs multiple times, once for each physical library branch. Some of the problems that need to be addressed are how to represent multiple occurrences of these sub-systems with respect to the rest of the system and with respect to other occurrences of the sub-system without representing a fixed number of multiple instances or specifying multiple occurrences redundantly. Once this is completed, future research in the application of formalized SA specifications would be greatly facilitated by the development of a CASE tool to construct and analyze formalized SA specifications.

The next two major units of work involve the application of formalized SA specifications in two ways:

1. Design of software systems from SA specifications is ad hoc at best. The transformation of an informal SA specification into a structure design (SD) is often a "magical" process. In many cases the specifications are thrown out and system design begins from scratch with little or no benefit derived from the specification phase. However, formalized SA specifications can be mathematically analyzed and measured. Design techniques based on correctness preserving transformations of SA specifications into a working implementation are more readily possible with mathematically precise specifications. Formalized SA specifications could be enhanced with attributes for code or language requirements so that an SA specification actually evolves into a SD.

Additionally, heuristics may be derived which can analyze SA specifications for requisite concurrency, early detection of deadlock, or other potential development problems. Restructuring heuristics may be derived for releveling the DFD hierarchy based on path length from stores, distribution of implementation processes across consumable flows, concurrent execution of implementation
processes across persistent flows, and implementation of certain SA components in hardware based on the appearance of analog flows in the specification.

2. Client interpretation of SA specifications takes on a whole new meaning when clients can execute the specification. Given the mathematically precise interpretation of formalized SA specifications, it should be possible to semi-automatically produce a working animation, i.e. prototype, of an SA specification [37, 57]. Clearly the software client can determine if his needs will be satisfied by experimenting with a prototype based on the formalized SA specification. Additionally, the prototype can be used as a test oracle to ensure that the finished product satisfies the clients requirements as characterized by the specifications.

A proposal to study the feasibility of semi-automatic prototyping from formalized specifications has already been submitted for approval. This research will develop and analyze techniques for helping to automate the production of a working prototype from formalized SA specifications. In particular, the research proposes to use the multi-tasking features of the programming language Ada to map a formalized SA specification directly to a working model of the system. The results of this research will:

(a) Lend insight into the automated production of a working prototype from formalized SA specifications.

(b) Suggest problems that need to be addressed by research into the development of design techniques for use with SA specifications.

The DoD requires the use of Ada and will often also require the use of SA
specification techniques. Ada is also well-suited to the implementation of concurrent/real-time systems [1, 4, 23]. Because of the large cost of software systems development, any private industries or government agency required or interested in using SA specification techniques to produce software systems in Ada would likely derive benefit from this work.

10.3 Benefits

Researchers are becoming increasingly convinced that more formal specification and design techniques represent:

1. an opportunity to continue the trend toward development methods that facilitate systematic analysis and review earlier in the life cycle, thus reducing error detection and correction costs [8, 10, 34, 57]; and

2. a basis for significantly enhanced automated analysis and support tools [9, 43, 48, 55].

It is simply too easy to suppress difficult issues during the specification phase of the software life cycle when one uses informal techniques. How do I know when I am done specifying? What do we do about duplicate record keys? These questions are too often unresolved with traditional, informal specification techniques. We can do a better job of wrestling with such issues earlier in the development cycle.

Formalized SA specifications will facilitate numerous automated tools. The following lists representative possibilities:

1. development of automated analysis tools. These analysis tools would be capable of providing software engineers with valuable information about the structure
and complexity of their SA specifications [15, 36, 40, 48].

2. support for systematic methods. More systematic methods will allow increased efficiency in the production of software systems from SA specifications [23, 25, 31].

3. automated concurrent system analysis, e.g., deadlock detection [40, 48].

4. semi-automated library retrieval. Software reuse is an often-used term that is difficult to realize in practice [1]. One major obstacle to successful reusability is that as software libraries grow in size, it is difficult for individual developers or teams to know what is available in a large library. (Without large libraries, it is impossible to foster extensive software reuse.) Using formalized SA specifications, it may be possible to get intelligent automated help in determining which library modules might be reusable in a new system.

5. real prototyping capabilities [37]. All too often “rapid prototyping” means “poorly implement only part of the system.” With more formal SA specifications — combining executable specification techniques and “canned” implementation strategies for commonly used abstractions, it will be possible to actually display full system behavior at the end of the specification phase. (Note that such prototypes may run slowly, and hence will not, in general, serve for evaluating conformance with performance requirements.)

6. document maintenance environments. It is currently expensive and difficult to maintain consistency between the various documents produced as part of a software project. Given more formal SA specifications, it will be possible to
provide automated support to recognize that a particular part of a specification is related to a particular part of a design or implementation document.

7. consistency checking. Researchers at Iowa State University are conducting research on automated consistency checking between specifications written in the SPECS language and implementations written in Pascal [8]. Note that this work is relating abstract specifications written over mathematical types like set and sequence to actual imperative language programs.

None of these tools would be feasible were it not for the advent of formalized SA specifications. The industry will be driven to these specification techniques by the potential benefits. "Specifications in Prolog" and "Prototyping in Prolog" are just early indications of this trend toward increased formalism earlier in the life cycle.
REFERENCES


