

Synthesizing Structured Analysis and Object-Oriented Specifications

David L. Coleman and Albert L. Baker

TR #94-04

March 1994

Categories and Subject Descriptors:

D.2.1 [**Software Engineering**] Requirements/Specifications

languages, methodologies;

D.2.2 [**Software Engineering**] Tools and Techniques

computer-aided software engineering (CASE);

D.4.7 [**Operating Systems**] Organization and Design

distributed systems

Additional Key Words and Phrases:

distributed systems specifications, formalized structured analysis, data flow diagrams

Submitted to *IEEE Transactions on Software Engineering*

© David L. Coleman and Albert L. Baker, 1994. All rights reserved.

Synthesizing Structured Analysis and Object-Oriented Specifications

David L. Coleman and Albert L. Baker

March 3, 1994

Abstract

Structured Analysis (SA) is a widely-used software development method. SA specifications are based on Data Flow Diagrams (DFD's), Data Dictionaries (DD's) and data transformation specifications (P-Specs). As used in practice, SA specifications are not formal. Seemingly orthogonal approaches to specifications are those using formal, object-oriented, model-based specification languages, e.g., VDM, Z, Larch/C++ and SPECS. These languages support object-oriented software development in that they are designed to specify abstract data types (ADT's). We suggest formalizing SA specifications by: (i) formally specifying flow value types as ADT's in DD's, (ii) formally specifying P-Specs using both the assertional style of the aforementioned specification languages and ADT operations defined in DD's, and (iii) adopting a formal semantics for DFD "execution steps".

The resulting formalized SA specifications, DFD-SPECS, are well-suited to the specification of distributed or concurrent systems. We provide an example DFD-SPEC for a client-server system with a replicated server. When synthesized with our recent results in the direct execution of formal, model-based specifications, DFD-SPECS will also support the direct execution of specifications of concurrent or distributed systems.

1 Introduction

Specifications define systems. They are the first document produced on a project which is intended primarily for technical designers and programmers. Specifications define the functionality of a system, as opposed to designs, which describe implementation data structures and program units. A specification technique should at least support a precise description of system functionality without requiring resolution of design decisions.

There are two de-facto camps in the Software Engineering community: advocates of formal specification techniques and advocates of informal techniques. The term “formal specification” refers to a mathematically precise definition of software functionality. In general use, informal specifications and formal specifications are mutually exclusive.

This mutual exclusion is, in part, the result of the disdain between industrial users of informal specification techniques and advocates of formal specification techniques. Arguments against the use of formal specifications include:

- The lack of evidence that formal techniques can be cost-effectively applied to real systems, as opposed to the simple examples presented in texts and papers on formal methods.
- The frequent role of a specification as a contractual agreement between clients and suppliers, which necessitates that specifications be comprehensible by diverse individuals not trained in formal specification techniques, e.g., managers and users.
- The use of formal specifications requires rigorous mathematical training.

Advocates of formal methods counter that software systems are inherently complex. Thus any precise definition of system functionality must rely on a formal, i.e., mathematical, definition of functionality. Furthermore, formal specifications readily lend themselves to more rigorous analysis, thus providing valuable information about the structure and complexity of proposed systems. Other touted benefits of formal methods are:

- Automated processing of specifications to assist in software development and verification of implementations.
- Executable specifications.

In general, formal specifications offer the opportunity to be more precise earlier in the development process.

In this paper, we present a formal specification technique, DFD-SPECS, which we have developed, used and taught over the last few years. DFD-SPECS is a formal method that is well-suited to the specification of distributed or concurrent systems. It is essentially a synthesis of Structured Analysis (SA) and formal Object-Oriented (OO) specification techniques.

SA specifications are a commonly used, graphical technique for describing system architecture [8, 26]. SA specifications are often touted as easily comprehensible to both users and technicians. The graphical representations provide convenient high-level views of system functionality. However, the underlying functionality of individual graphic elements, as well as the specific semantics of the element interfaces, are not rigorously defined [15, 7, 6]. Thus a traditional SA specification is not formal.

SPECS [1] is a formal, model-based specification language analogous to Z [11], VDM [13] and Larch/C++ [5, 18, 17]. Model-based specifications use discrete mathematical structures to model the state of a system and to define the operations that transform system state. Model-based specification methods are well-suited to the specification of Abstract Data Types (ADT's). Hence they foster an object-oriented approach to software development. The abstract domain of an ADT defines the set of states the system may be in. The abstract operations define how the system can be transformed from one state to another. SPECS uses this same model-based approach to specify ADT domains and operations. An example use of SPECS is provided in Section 3.

Most widely-known specification techniques, both formal and informal, do not readily lend themselves to the specification of distributed or concurrent systems. Those practical techniques that do claim to specify distributed or concurrent systems typically operate at a design level,

e.g., requiring a master–slave process architecture [24]. While this design level perspective must subsequently be addressed, specifications should be able to represent potential for concurrency independent of these design-level concerns.¹

Existing formal specification techniques for distributed or concurrent systems are often text-based and incorporate forms of temporal logic [21]. While such techniques provide the requisite rigor, they may require a level of mathematical sophistication that is simply not cost-effectively achieved in production software development environments.

There are some graphics-based specification techniques for distributed or concurrent systems, several of which are variations on petri nets [20, 25]. Petri net specifications can be mathematically analyzed for synchronization issues like deadlock, liveness, etc., [19]. However, petri net specifications generally lack a representation for functionality. They address only synchronization issues and do not describe the explicit functionality of a system. DFD-SPECS provides a model of concurrency similar to petri nets, but also supports the precise specification of system functionality. Thus specifications in DFD-SPECS are graphics-based and can be analyzed mathematically.

We have derived DFD-SPECS by starting with the basic features used in traditional SA specifications (flows, bubbles (processes), stores and terminators) and providing a more formal semantics for these features. With respect to flows, we have resolved issues like formal abstract data type (ADT) specifications for flow values, continuous vs. discrete flows, flow convergence, flow divergence, and balancing flows across different levels of decomposition. Our initial results are contained in [6].

In this paper, we focus on our results pertaining to “what bubbles mean”. (However, this requires resolution of some of the issues pertaining to flows enumerated in the preceding paragraph.) Bubbles in DFD-SPECS actually warrant the label “process”, as opposed to “procedure”. We have developed a semantic notion of what it means for a DFD process to execute, and an assertional

¹In [3], Banâtre and Métayer make the distinction between *logical parallelism*, the possibility of describing a program as a composition of several independent tasks, and *physical parallelism*, the distribution of tasks on several processors.

approach to defining *when* a process executes and *what* “values” are produced. Thus we have a formal, assertional approach to what in traditional SA are called P-Specs.

In Section 2, we provide a brief and informal overview of the syntax and semantics of traditional SA specifications. The model-based specification language we use to define ADT’s, SPECS, is described in Section 3. We use SPECS to specify “flow values” in DFD’s. DFD-SPECS is defined in Section 4. We pay particular attention to the features of DFD-SPECS that pertain to processes and the “execution” of DFD-SPECS.

We then provide an example DFD-SPEC for a replicated data server in Section 5. The main point of the paper is not the particular approach to replicated data servers—we only handle two servers—but we are confident the reader will see merit in the approach to specifying this problem. In Section 6, we discuss the implications of DFD-SPECS and future areas of research.

2 Traditional Structured Analysis

Traditional SA uses three modeling tools to specify systems:

1. Data Flow Diagrams (DFD’s)
2. Data Dictionaries (DD’s)
3. Process Specifications (P-Specs)

We discuss each of these modeling tools in the following subsections.

2.1 Data Flow Diagrams

DFD’s provide a static, graphical representation of system functionality and are the main SA modeling tool. A DFD is data-focused, as opposed to procedural-focused, and represents information flow in systems. A DFD depicts the source of each process’s information and the destination of

each process's output. In this section we adopt the graphical and syntactic notation for DFD's used by De Marco [8] and Yourdon [26]. Figure 1 provides a simple DFD.²

The four components of a DFD are:

1. Bubbles, represented by circles with an associated P-Spec (or DFD refinement, if dealing with hierarchical DFD's),
2. Flows, represented by directed edges with an associated DD entry (DDE),
3. Stores, represented by parallel lines (boxes with open ends) with an associated DDE,
4. Terminators, represented by rectangles.

Each instance of a component is uniquely labeled.

We can now specifically note why traditional DFD's are not formal specifications:

1. Bubbles are not usually specified formally. Their P-Specs are often represented using an algorithmic pseudo-code, or other algorithmic representation.
2. There is no notion of DFD "behavior". There is no notion of when "bubbles get to work" and when they "generate output".
3. Flows in traditional DFD's are usually typed, but there is no notion of values (tokens) "on the flows", and hence no model for the actual movement of values in a traditional DFD.

Bubbles represent data transformations in systems. In the traditional SA literature, bubbles are often referred to as "processes". This is misleading, because of the connotation of the term for distributed or concurrent systems. System design from a SA specification typically involves "analyzing" the DFD so that each bubble may be implemented as one or more imperative language

²Traditional SA requires the example in Figure 1 to be represented as a hierarchy of DFD's in which the top level, *Level-0*, consists of a single "bubble" representing the entire system, as well as the flows into, and out of, the system. Subsequent levels in the hierarchy represent decompositions of bubbles in the preceding level. However, the example as presented depicts the aspects of DFD's that are pertinent to this paper.

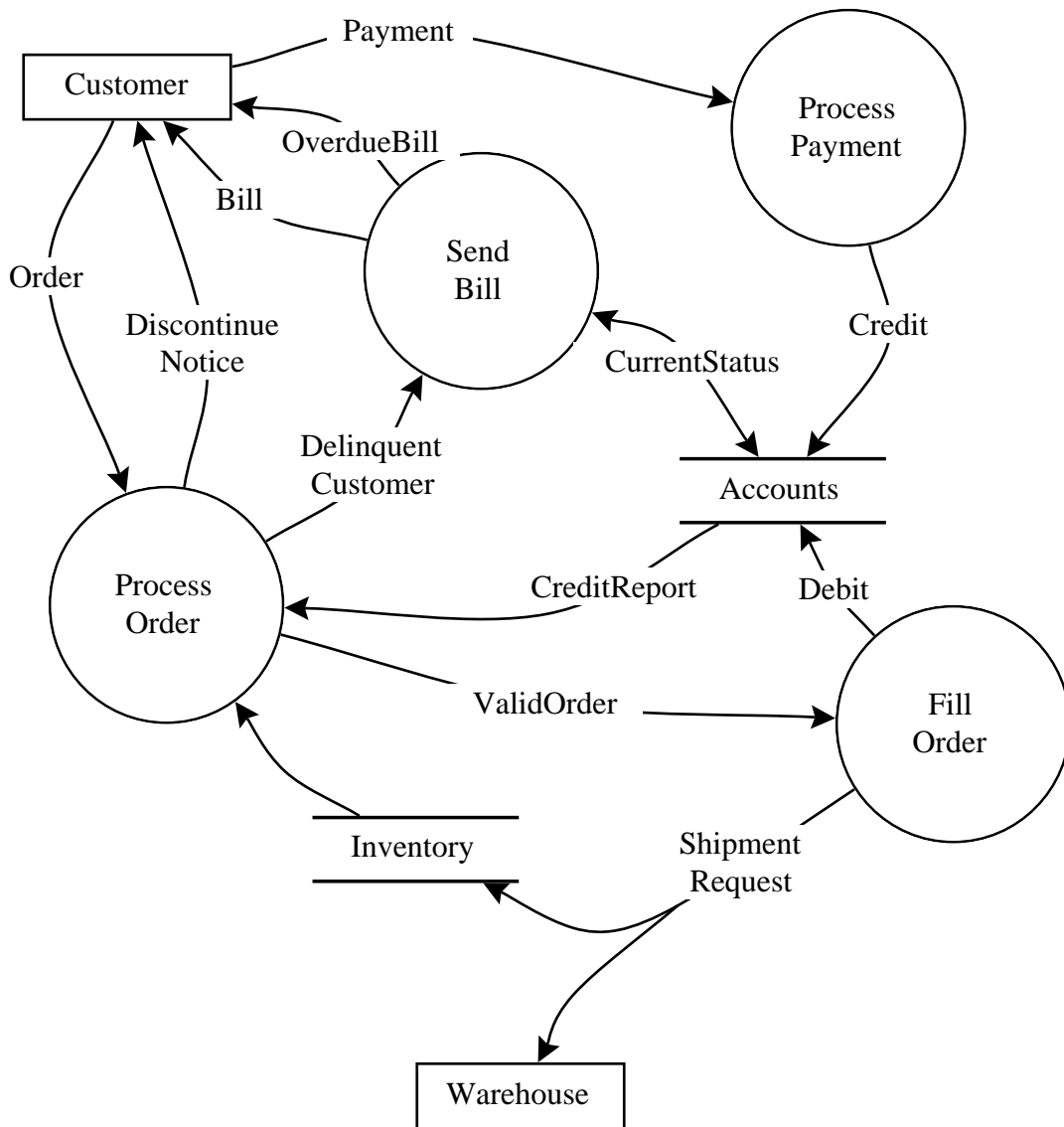


Figure 1: Example DFD of an Order Processing System

procedures or functions. Hence our use of the term *process bubble* or just *bubble* to refer to the “circles” in traditional DFD’s. Even *real-time* extensions to SA do not clearly distinguish bubbles as procedures from bubbles as true processes. In Section 4, we provide a more rigorous and abstract form of P-Specs and a meaning of DFD execution that is independent of a particular design and implementation strategy.

Flows define the information pathways in systems. Flows may 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. Flows represent the movement of data from one DFD component to another. Thus flows into bubbles, *in-flows*, and out of bubbles, *out-flows*, represent the data inputs and outputs of the computation represented by the bubble.

However, flows in traditional DFD’s are only a static representation of system information flow. These flows do not model data value movement from a producer to a consumer. Is data explicitly prompted for, or does it arrive independent of any action on the part of the consumer? In Section 4, we define three flow value behaviors that characterize the dynamic movement of data through the system.

The label attached to a flow should describe the data that can move on this particular path. This definition of data is provided by a DDE for each flow label. Traditionally, each DDE provides a syntactic representation of the domain of values for a flow. However, within traditional P-Specs, each flow label is also used as a variable to reference the data values represented by the flow. Thus flow labels have played the role of both variable names and type names. This has meant, often to comply with imposed standards, that developers have adopted the “flow label as variable name” convention. In Section 4, we provide a flow labeling scheme that eliminates this redundancy and still allows each flow to be uniquely identified by a label.

Flows in traditional DFD’s can also be shown as diverging, converging, or with arrowheads on both ends, as in Figure 2. There is potential ambiguity in the interpretation of these structures. For example, a *divergence* might represent that duplicate copies of a data value, or *token*, are being

sent to different destinations, that a composite token is being split into its constituent values, or that a flow carries different token types that are being directed differently. There are analogous ambiguities in the interpretation of convergences and dialogue flows.

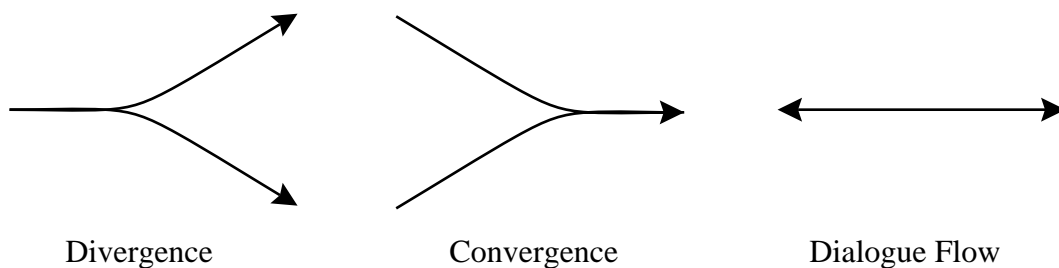


Figure 2: Diverging, Converging, and Dialogue Flows

[6] contains a resolution to these semantic ambiguities in these flow constructs. The resolution of these ambiguities is based on labeling flows with both flow names and types and providing formal specifications of the flow types in the DDE's. It turns out that a precise resolution of the semantics of flow convergences and divergences is pertinent to a formal semantics for *flow balancing*, which is the association of flows across different levels in a hierarchical DFD.

We do not cover these results in this paper, and hence have limited ourselves to example DFD's that do not contain flow divergences or convergences. In this paper, we consider a dialogue flow as an abstraction of two flows with identical type information “running in opposite directions”. Their main purpose is to unclutter a diagram.

Stores are another source of ambiguity in DFD's. The traditional view of a store in a DFD is that it represents collections of data that the system must remember for some period of time. In practice, stores are often just thinly veiled abstractions for files or databases. The term “file” certainly implies a very strict implementation mentality and one has to wonder whether this is the appropriate level of abstraction for components in a specification.

There are several pertinent questions pertaining to the semantics of stores:

1. How do stores represent data?

2. When, and by what stimulus, do stores produce values on their out-flows?
3. When, and by what mechanism, do stores receive values on their in-flows?
4. Do stores process data like a bubble or are they in some sense more passive?

Again, [6] contains a resolution to these questions concerning stores. We suggest that stores are not “primitive” in the same sense as bubbles and flows in that stores can be modeled as DFD’s that contain only bubbles and flows. Since we do not cover the semantics of stores in this paper, we have limited our example DFD’s to those containing only bubbles, flows and terminators.

Terminators represent the external entities with which the system communicates. Terminators are often called *sources* and *sinks* for information outside the control of the system. A terminator is identified by its label. The behavior of terminators is not defined in traditional DFD’s. We use some of our facilities for P-Specs to provide partial specifications of the time-dependent behavior of terminators. Some modeling of terminator behavior is necessary for validation of DFD-SPECS, whether via rigorous analysis or direct prototyping.

2.2 Data Dictionary (DD)

The traditional DD modeling tool provides syntactic descriptions of the data represented by flows and stores. For example, the flow labeled Payment in Figure 1 might be defined as follows:

$$\text{Payment} = \text{Customer} + \text{OrderNumber} + \text{Amount}$$

Each subordinate of this definition must also be defined in the DD. For example, OrderNumber might be defined as follows:

$$\begin{aligned} \text{OrderNumber} = & \text{StateCode} + \text{AccountNumber} \\ & + \text{SalesmanID} + \text{SequentialCount} \end{aligned}$$

In traditional SA, a Data Dictionary Entry (DDE) must be provided for every flow label, store label, and all subordinate labels used to define them. A label whose DDE is empty is called an *elementary data element* and is said to be “self-defining”. We guess this puts inordinate stress on

the flow label mnemonics. Each non-empty DDE is defined by an expression whose syntax and informal semantics is analogous to a BNF description of the “language” of legal values. This lexical description is based on undefined primitives (elementary data elements) and character strings. Furthermore, operations on flow and store data elements are limited to those described in P-Spec pseudo-code. There is no commonly accepted and general set of operations on these syntactic entities that support the precise specification of the functional behavior of bubbles.

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 Section 4, we propose that DDE’s consist of specifications of abstract data types using formal, model-based, object-oriented specifications.

2.3 Process Specifications (P-Specs)

P-Specs, also referred to as “mini-specs”, should provide a specification of bubble behavior. Thus an individual P-Spec should define the relation of out-flow values to in-flow values. Various representations of this functionality have appeared in the traditional SA literature, including decision tables and decision trees, structured English (pseudo code), flowcharts and Nassi–Shneiderman diagrams, and even informal English [9]. The first three methods represent those traditionally used by De Marco [8]. Use of these techniques result in either ambiguity or over specification—in the sense that the specification contains data structure design and implementation details.

In Section 4, we take an assertional and model-based approach to P-Specs, thereby integrating current object-oriented approaches to specification with these SA techniques.

2.4 Hierarchical (Leveled) Data Flow Diagrams

A DFD provides a system overview, but for complex systems the number of bubbles appearing in a DFD becomes unmanageable. Leveled DFD’s allow for abstraction of groups of DFD elements at successively higher levels. When a particular DFD becomes too large, we can partition the elements 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 DFD's.

Each bubble that decomposes into a DFD represents an abstraction of its decomposition. To make this hierarchy complete, each bubble in a DFD must be represented by either a P-Spec or another DFD at a lower level in the hierarchy. Bubbles specified using P-Specs are called *primitive bubbles*. In fact, the more formal semantics we provide for DFD's requires that we view a DFD as a network of primitive bubbles. Thus, for the purpose of this paper, we deal with DFD's containing only primitive bubbles. See [6] for further discussion of the issues in providing a more formal semantics for leveled DFD's.

3 SPECS

SPECS is a model-based language for specifying ADT's. It was developed at Iowa State University as a research tool and for use in Software Engineering courses at the graduate and undergraduate level. In SPECS, as with other model-based specification languages, ADT's are specified by first composing an abstract domain from primitive abstract types and then defining a set of "client-callable" operations over that domain.

We will use SPECS in two ways in this paper. The most important use is in DDE's for the flows in our example DFD-SPEC in Section 5. For each flow type appearing in an example, we provide a specification of that type in SPECS. Thus each flow type will be precisely specified, and we gain the additional expressive advantage of being able to use the operations over the types in P-Specs.

The second use of SPECS in the paper is as a higher level abstraction of our example DFD-SPEC in Section 5. The example DFD-SPEC is for a client-server application. The clients in this example can view "the services provided" as a simple ADT Table. This ADT Table is the example we use in this section to introduce SPECS. A complete definition of SPECS is found in [1, 2].

3.1 An Example ADT

The ADT below illustrates the syntax and semantics of SPECS. Note the two main sections of the specification of ADT Table: domain and operations. An additional definitions section can be used to define abstract constants, provide additional abstract type declarations that do not arise naturally in the composition of the domain of the ADT, and expression definitions which are used to “modularize” other assertions in the ADT specification. The simple ADT Table does not require a definitions section. A general discussion of each of these sections follows the example.

adt:Table

domain

source set

Table = **set of** EntryType;
EntryType = **tuple** (*Index*:Domain, *Value*:Range);
Domain = **generic**;
Range = **generic**;

invariant

(* Elements of a Table must have unique indices. *)
 $\forall(T:\text{Table})[$
 $\forall x \forall y [x \in T \wedge y \in T \wedge x \neq y \Rightarrow \text{Index}(x) \neq \text{Index}(y)]]$

operations

function NewTable:Table;

post: NewTable = { }

function WriteEntry(*T*:Table; *I*:Domain; *V*:Range):Table;

post: WriteEntry = $(T - \{e \mid e \in T \wedge \text{Index}(e) = I\}) \cup \{(I, V):\text{EntryType}\}$

function ReadEntry(*T*:Table; *I*:Domain):Pair;

(* Pair is a simple ADT presented in Section 5. *)

post: $\exists e [e \in T \wedge \text{Index}(e) = I \wedge \text{ReadEntry} = \text{MakePair}(I, \text{Value}(e))]$
 $\vee \neg \exists e [e \in T \wedge \text{Index}(e) = I \wedge \text{ReadEntry} = \text{MakePair}(I, \text{Undefined})]$

3.2 ADT Domain

The domain section of a SPECS ADT provides a discrete mathematical model of the set of values for the ADT. The domain specification has two parts. The source set is just the declaration of all the abstract types used in the ADT domain specification, including one declaration for the ADT type name. Thus the source set specifies the discrete mathematical structure of the “model” of the ADT values. The abstract type declarations in the source set section are based on the intrinsic structured primitive types (set, sequence, tuple, alternatively defined types, and recursively defined types) and on a collection of intrinsic simple primitive types (integer, real, char, string, enumerated types, and the type generic—to allow for polymorphic ADT’s in the usual manner). Associated with each of the intrinsic types is a set of operations (e.g., set union, \cup , for the intrinsic structured primitive type set) which we use in writing assertions.

We present the composition of these abstract type name declarations in a top-down manner. Thus in the Table example the type Table is declared first. In this example, our model of values in the domain of the ADT Table is just a set of pairs, where each pair has an Index component and a Value component.

The invariant part of the domain section is just an assertion used to otherwise limit the set of abstract values in the domain. In our example, the invariant stipulates that in any given instance of the ADT Table, each entry in the Table must have a unique Index component.

3.3 ADT Definitions

The definitions section supports the definition of identifiers not defined elsewhere in the ADT.

Constant definitions may be generalized to any constant that satisfies a particular first order predicate using the keywords **any** and **such that**. For example, the following constant declaration defines `MaxListLength` as any positive integer.

$$\text{MaxListLength} = \mathbf{any:integer\ such\ that\ MaxListLength} > 0;$$

This constant could be used in the specification of an ADT List by including the following expression within the domain invariant.

$$\text{length}(L) \leq \text{MaxListLength}$$

Constant values may also be explicitly stated. For example:

$$\text{MaxStackSize} = 10:\text{integer};$$

Conventions are available for declaring constants of any structured or primitive type.

Type definitions provide data types for operation parameters, operation result types, expression definitions, and expression definition parameters. In other words, the types portion of the definitions section contains declarations of all abstract types which do not arise in the composition of the ADT source set, but which are otherwise used in the ADT specification. These type declarations have the same format as the source set declarations.

Expression definitions provide for parameterized named expressions that can subsequently be used in other assertions. The general syntax of an expression definition is:

define *ExpressionName*(P_1 :ParameterType₁,... P_n :ParameterType_n) **as** ResultType
such that $Q(\text{ExpressionName}(P_1, \dots, P_n), P_1, \dots, P_n)$

Q represents a first order predicate in which “*ExpressionName*(P_1, \dots, P_n)” is defined in terms of P_1 through P_n as an instance of a ResultType object.

3.4 ADT Operations

The specified ADT operations provide the only means by which a client of the ADT can create, modify, or query an instance of the ADT. An operation is defined by providing a procedure or function header, (in this paper we use Ada-like formal parameters), a precondition and a postcondition.

The precondition is a first order predicate assertion over the set of **in** parameters.³ In the case where there is no precondition, i.e., the precondition is just the constant value *true*—the

³All function parameters are implicitly defined as **in** parameters and the function name is the sole **out** “parameter.”

precondition (including the keyword **pre:**) can be omitted. The postcondition is a first order predicate assertion which defines the **out** parameters in terms of the **in** parameters. When a procedure's parameter is used for both input and output (**in-out**), the use of the parameter name in the postcondition refers to its input value. The output (post-state) value of an **in-out** parameter may be referred to in the postcondition by priming (') the parameter name.

The ADT Table has a fairly simple set of operations: an initial constructor `NewTable`, an operation `WriteEntry` that allows a client⁴ to add a new entry to the table or to overwrite the value of an existing entry, and an operation `ReadEntry` that allows a client to find out if a particular index value I is the index of an entry in a table T , and if it is, the associated value. This is all we need for the example DFD-SPEC in section 5, but one can easily envisage other operations, e.g., `RemoveEntry`, `ClearTable`, to make a more useful ADT. The important point about the ADT Table as it is used in Section 5 is that it provides the only perspective clients need for accessing the replicated data base.

4 DFD-SPECS

DFD-SPECS combines the formal, object-oriented specifications of SPECS with a formalized variation of SA specifications. DFD-SPECS are no less precise than SPECS. DFD-SPECS incorporate SPECS for DDE's and formal assertional P-Specs. DFD-SPECS maintains the graphical nature of traditional SA specifications and has a semantics for an "execution" of DFD's. DFD-SPECS differs specifically from traditional SA specifications in the following ways:

1. A new flow labeling scheme is used to distinguish flow types from flow values and eliminate the need for redundant DDE's.
2. Flow types are defined using either SPECS ADT's or SPECS primitive types.

⁴of the ADT. We are using the term "client" in the same sense that we talk about *clients* of C++ classes.

3. Three types of flow value behaviors are identified and used to characterize the movement of values on flows.
4. DFD bubbles are interpreted as abstractions of true “processes” with all the connotations of the term “process” in distributed or concurrent systems.
5. P-Specs are expressed using a Mealy-like state machine with formal assertional specifications of synchronization and functionality.
6. Terminator temporal behavior is also defined using a Mealy-like state machine.
7. An operational semantics for DFD-SPECS is defined.

4.1 Flow Labels

We adopt a flow labeling syntax that does not require redundant definition of type structure, but does allow the distinction between two flows with the same type structure. In DFD-SPECS, a flow label is prefixed by an identifier separated from the usual flow label by a colon. Only the usual flow label (i.e., the suffix identifier) following the colon is defined structurally in the DD. Unique identification of flows is provided by the prefix identifier. Redundant definition of type structure is no longer necessary since distinct flows can share a common suffix. A merge process bubble might have two in-flows labeled “*A*:List” and “*B*:List” and the merge P-Spec can refer to the prefixes *A* and *B*. The flow label suffix List could be defined as a sequence in a DDE.

4.2 Flow Type Structures

In DFD-SPECS, we replace the traditional description of syntactic structure provided by the DDE expressions with the richer semantics of abstract model specifications using SPECS. To facilitate the formalized expression of P-Spec functionality, we define the type structure associated with the suffix of a flow label with either an ADT or a SPECS type definition. The set of operations provided by an ADT represent the only methods by which a P-Spec can construct, modify, or

query the value on a flow whose label suffix is defined by an ADT. This encapsulates the flow value in an object-oriented specification. Using these ADT operations we can write formal assertional specifications for P-Spec functionality.

To avoid further redundancy within the collection of DDE's used to define the type structure of flow values, we adopt the following scoping rule: any typename identifier used in an ADT must either be defined locally within the DDE containing that ADT or must be defined in another DDE—either as an ADT or a simple primitive type. A SPECS abstract constant definition may be encapsulated by a DDE, thus allowing the constant to be referenced in multiple ADT's or P-Specs. A formalization of stores, not provided in this paper, may contribute additional varieties of DDE's.

4.3 Flow Value Behaviors

Ward uses the terms *discrete* and *continuous* to describe flows [23]. A *discrete flow* may contain zero or more instances of a flow value at any point in time. Discrete flow values are *produced* by the flow's source and destructively *consumed* by the flow's destination. Thus discrete flows behave much like a time-ordered message channel in a distributed system. A *continuous flow* always contains exactly one flow value. The continuous flow value is destructively *written* by the flow's source and non-destructively *read* by the flow's destination. Thus continuous flows behave much like a shared variable in a concurrent system.

On the other hand, Hatley and Pirbhai also use the terms *discrete* and *continuous* to describe the way in which flow values change [10]. In their usage, each *discrete flow* describes a function mapping time to a set of discrete values. Each *continuous flow* describes a function mapping time to a continuous set of values. In SPECS the only true continuous domain is provided by the simple primitive type *real* and thus a continuous flow must only model real values.

From these two distinct uses of the terms *discrete* and *continuous* we derive three flow value behaviors; *analog*, *persistent*, and *consumable*. An *analog flow* combines Ward's continuous existence of single value with Hatley's continuous function over time. An analog flow can be used to characterize real world input such as time, temperature, etc, and must have type *real* in its associ-

ated DDE. A *persistent flow* combines Ward's continuous existence of a single value with Hatley's discrete function over time. A *consumable flow* combines Ward's discrete existence of zero or more flow values with Hatley's discrete function over time. A consumable flow is considered a FIFO queue of flow values. All flows provide reliable data movement. The analogies of a persistent flow with a shared variable and a consumable flow with a message channel are useful for comprehension, but do not dictate a particular implementation. In the example in Section 5, we use a dashed line to represent consumable flows and a solid line to represent persistent flows. Analog flows are not used in the example.

4.4 Process Bubbles

In a distributed or concurrent system the term *process* refers to a program that has started execution, but has not yet terminated. The differences between a process and an imperative procedure include:

1. A procedure is not always active (does not have self control) whereas a process is always active, although its execution may be suspended.
2. A process can decide on its own "when to compute". Doing so requires that it actively observe its environment. A procedure becomes active, executes, and terminates only when invoked by an external entity.
3. A procedure is usually provided with the same set of input parameters and produces the same set of output parameters. A Process may consume different inputs and produce different outputs depending on the process's execution state and its "activation rules" (as in 2. above).
4. A procedure does not (locally) remember results from previous activations. A process, since it may always be active, can remember previous results.

This is not necessarily a definitive list of differences, but does serve to distinguish the concepts.

Real-time extensions to SA do provide for the direct specification of when a DFD bubble can execute [23]. Ward provides two types of DFD bubbles; one a traditional “data transformation” bubble; the other a “control transformation.” Control transformations serve to activate and deactivate data transformations. Typically, only one control transformation may activate or deactivate a data transformation. Higher level control transformations activate and deactivate lower level control transformations. In effect, the control transformation hierarchy provides a representation of control structure similar to a calling chart for imperative programs, or a master-slave implementation architecture for concurrent or distributed systems.

Our research and research by Kung lead us to question whether control transformations are necessary for specifications [7, 15]. In DFD-SPECS, the specification of a process bubble provides self-determination of when a process bubble executes.

Kung developed an expression notation for representing when a process bubble could “do it’s work” and what outputs would be produced. Kung’s work did not include a specification of functionality—just of process timing. Kung used this notation to develop a consistency check between parent bubbles and child specifications (either DFD or P-Spec).

However, Kung’s method fell short by failing to model the state of a process. We extended Kung’s notation to provide a state-based specification of process timing. We also provided a consistency checking mechanism and acknowledged its limitations. We then added formal assertions to specify process functionality.

4.5 Assertional P-Specs

In DFD-SPECS, a bubble is an abstraction of a true concurrent process. P-Specs are expressed using a variation of state transition machines referred to as Mealy machines [12]. The behavior of a bubble is defined by the labeling of the state transitions in the P-Spec. Each transition labeling has the following form:

$$\langle \text{enabling-condition} \rangle : \langle \text{precondition} \rangle \mid \langle \text{postcondition} \rangle$$

The enabling-condition provides the specification of when a bubble may execute. When one or more enabling-conditions associated with transitions out of the current state are true, then the P-Spec is enabled. The enabling-condition is a SPECS first order assertion written over the bubble’s in-flows. The value of any analog in-flow or persistent in-flow is referenced by the flow label prefix. A consumable in-flow value may or may not exist at any particular point in a DFD execution. We extend SPECS assertion notation with two operators denoting existence and non-existence of a flow value on a consumable in-flow. Given a consumable in-flow with label prefix X , the boolean expression X^+ is true if and only if there exists a value on flow X , and $X^- \equiv \neg X^+$. When a flow value is present, the value is referenced using the flow label prefix X .

The precondition is an assertion over in-flows to a P-Spec defining any constraints on values of in-flows, where these precondition constraints are not properly part of the enabling condition. Thus P-Specs preconditions are analogous to the preconditions in an ADT operation. As with traditional SPECS preconditions, we drop the precondition component of a transition label (and preceding colon) when the precondition is just *true*.

The postcondition is an assertion over both in-flows and out-flows of a P-Spec defining the values produced by the P-Spec. Analog out-flows and persistent out-flows mentioned in the postcondition represent destructive “writes” to the out-flow value. Consumable out-flows mentioned in the postcondition represent “sends” of a new consumable flow value. At most one write or send on any out-flow is allowed per transition (firing of the process).⁵ The P-Spec defines only the minimum out-flow values needed to satisfy the postcondition. We use the \models operator, read *produces*, since satisfaction of the enabling-condition and precondition produces out-flow values which satisfy the postcondition.

Several syntactic representations of P-Specs are possible; state transition diagrams with appropriately labeled transitions, tabular representations, a textual list of transition labels with initial and final states, etc. In fact, state could be represented as a flow value on a *self-looping flow*, but

⁵Of course, the “value” sent may itself be a set or sequence of values.

we have found that the state machine perspective can significantly decrease the complexity of the textual enabling rules.

4.6 Terminators

Since terminators represent external entities with which the system interacts, they too can be modeled as true processes by a Mealy-machine. Since they are not part of the system, we may not be able to fully specify their functional behavior. (If one of our readers has a complete specification for the ubiquitous terminator USER, please send us a copy immediately.) However, modeling the temporal behavior of terminators does allow us to check the consistency of the system's temporal interaction with its external environment [7].

In the case where a terminator represents an external system or device, such as a file system, then the functional behavior of the terminator is just the specification of that system. Thus we do not preclude the complete functional specification of terminators.

4.7 Operational Semantics

The formalization of P-Spec's (and terminator's) temporal behavior and the formal interpretations of flow behavior form the basis for an operational semantics of a DFD-SPEC. This operational semantics, or *execution*, of a DFD-SPEC is derived from the *enabling* and *firing* of processes.

A two step firing rule is used to model computation [16]. Each process is either in a working meta-state or an idle meta-state. In an idle meta-state, the process is monitoring its in-flows and waiting to become enabled. In a working meta-state, the process has already consumed its enabling in-flows and has yet to produce its out-flows. Processes in the working meta-state are always enabled.

The execution of a DFD-SPEC is a sequence of firings of enabled processes. At each step a non-deterministic choice is made from the set of all enabled processes. If the chosen process was idle, the process consumes all consumable in-flow values (and references the current value of analog and persistent in-flows) that are referenced in the enabling transition label. The process is then placed

in the working meta-state. If the process was in the working meta-state, the process produces out-flow values which satisfy the postcondition of the enabling transition label applied when the process moved to the working meta-state. The out-flow values produced propagate through the DFD according to the semantics of consumable and persistent flows described earlier.

This execution interpretation is similar to the execution of a petri net [20]. Thus DFD-SPECS are well suited to the specification of the synchronization primitives of concurrent or distributed systems [25].

5 A Replicated Server Example

Recall the ADT Table from Section 3. In this section we specify a distributed implementation of Table using two servers. We assume processes do not fail and have not specified system error recovery.

Each client views the system as an implementation of the ADT Table. Clients can write entries and read entries. Each server locally maintains the minimum information needed to satisfy its clients' requests. When necessary, a server will exchange information with the other server to update its local information. Figure 3 provides the DFD overview of the replicated server system. The two servers are the process bubbles labeled Alpha and Beta. Each server's clients are represented by a single terminator.

Terminator functionality is left essentially unspecified, other than the type information already given for their in-flows and out-flows, and the fact that the semantics of DFD-SPECS does insure that servers will receive terminator messages on a given flow in the order in which they were generated. We don't specify the actual number of clients that talk to a particular server. We also do not provide any specification of the client terminators to insure terminators won't generate race conditions by incessantly generating server requests while ignoring server responses. Just to insure the terminators can be treated in the same manner as the process bubbles, we assume there is only one state for terminators, with the following transitions from this unique state to itself:

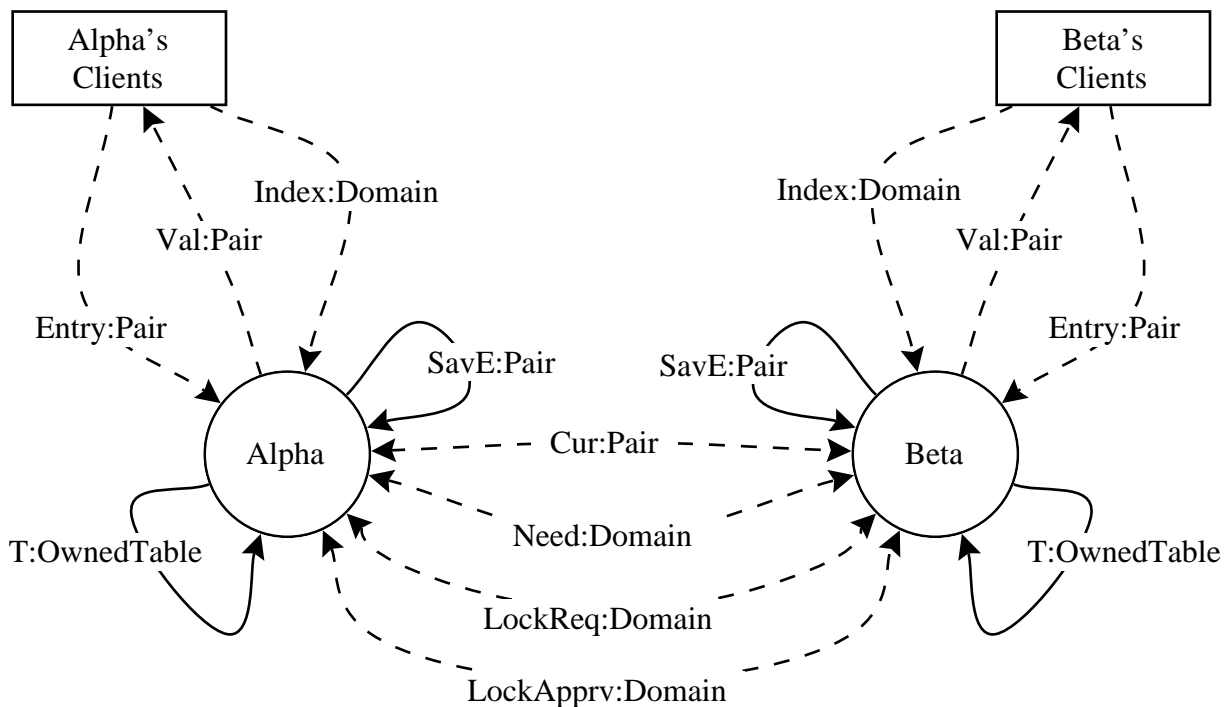


Figure 3: Context Data Flow Diagram for Two Server System

- #1 $\text{true} \models \text{Index}$ —client sends a Domain value on flow Index, corresponding to an invocation of the function ReadEntry.
- #2 $\text{Val}^+ \models \text{true}$ —client receives a Pair value on flow Val, corresponding to a return value of the function ReadEntry.
- #3 $\text{true} \models \text{Entry}$ —client sends a Pair value on flow Entry, corresponding to an invocation of the function WriteEntry.

The DD contains four entries: 2 ADT's, OwnedTable and Pair, and 2 types Domain and ExtendedRange. OwnedTable models each server's local copy of the table information. OwnedTable is similar to the ADT Table except each entry in the table has a list of owners. If a server owns an entry, the server knows that entry is current. To write an entry in response to a client's request, the

server must gain sole ownership of the entry. Each server has its own instance of OwnedTable—represented by a self-looping flow, T:OwnedTable, in Figure 3.

adt:OwnedTable

domain

source set

OwnedTable = **set of** OwnedEntryType;
 OwnedEntryType = **tuple** (*Index*:Domain,
 Value:ExtendedRange,
 Owners:**sequence of** ProcessId);
 ProcessId = (*Alpha*, *Beta*); (* An enumerated type. *)

invariant

$\forall(T:\text{OwnedTable})[$
 (* Elements of a table must have unique indices. *)
 $\forall x \forall y [x \in T \wedge y \in T \wedge x \neq y \Rightarrow \text{Index}(x) \neq \text{Index}(y)]$
 (* Every entry has at least one owner and any owner appears at most once in the owners
 sequence. *)
 $\wedge \forall x [x \in T \Rightarrow \text{Owners}(x) \neq \langle \rangle$
 $\quad \wedge \forall i \forall j [1 \leq i < j \leq \text{length}(\text{Owners}(x)) \Rightarrow \text{Owners}(x)_i \neq \text{Owners}(x)_j]]]$
 (* $\langle \rangle$ denotes the empty sequence. *length* and subscripting, e.g., $\text{Owners}(x)_i$, are operations on the primitive structured type sequence. Subscripting returns the i^{th} element of the sequence. *)

definitions

expressions

(* Remove defines the sequence S with the first occurrence of process Id P removed. *first* and *tail* are sequence operations that return the first element of the sequence and the sequence with the first element removed, respectively. \parallel is sequence concatenation. *)

define $\text{Remove}(S:\text{sequence of ProcessId}; P:\text{ProcessId})$

as sequence of ProcessId such that

$(S = \langle \rangle \Rightarrow \text{Remove}(S, P) = \langle \rangle)$

$\wedge (S \neq \langle \rangle \Rightarrow (\text{first}(S) = P \Rightarrow \text{Remove}(S, P) = \text{tail}(S)))$

$\quad \wedge (\text{first}(S) \neq P \Rightarrow \text{Remove}(S, P) = \langle \text{first}(S) \rangle \parallel \text{Remove}(\text{tail}(S), P))$

(* OtherOwners defines the sequence of all owners, except process Id P , of an entry whose index is I . *)

define $OtherOwners(T:OwnedTable; I:Domain; P:ProcessId)$

as sequence of ProcessId such that

$\neg \exists e[e \in T \wedge Index(e) = I \wedge OtherOwners(T, I, P) = \langle \rangle]$

$\vee \exists e[e \in T \wedge Index(e) = I \wedge OtherOwners(T, I, P) = Remove(Owners(e), P)]$

operations

function $NewOwnedTable:OwnedTable;$

post: $NewOwnedTable = \{\}$

(* AdmitEntry either adds a new entry to the table with owner P or it modifies an existing entry and adds P to its owners list. *)

function $AdmitEntry(T:OwnedTable; I:Domain; V:ExtendedRange; P:ProcessId):OwnedTable;$

post: $AdmitEntry = (T - \{e \mid e \in T \wedge Index(e) = I\})$

$\cup \{(I, V, \langle P \rangle \parallel OtherOwners(T, I, P)) : OwnedEntryType\}$

(* AddOwner adds P to the owners list of an existing entry. *)

function $AddOwner(T:OwnedTable; I:Domain; P:ProcessId):OwnedTable;$

pre: $\exists e[e \in T \wedge Index(e) = I]$

post: $\exists e[e \in T \wedge Index(e) = I]$

$\wedge AddOwner = (Index(e), Value(e),$

$\langle P \rangle \parallel OtherOwners(T, I, P)) : OwnedEntryType]$

(* RemoveOwner removes P from the owners list of an existing entry. *)

function $RemoveOwner(T:OwnedTable; I:Domain; P:ProcessId):OwnedTable;$

pre: $\exists e[e \in T \wedge Index(e) = I \wedge \langle P \rangle \neq Owners(e)]$

post: $\exists e[e \in T \wedge Index(e) = I]$

$\wedge RemoveOwner = (Index(e), Value(e),$

$Remove(Owners(e), P)) : OwnedEntryType]$

(* AccessTable returns the value associated with a particular index in the table. *)

function $AccessTable(T:OwnedTable; I:Domain):Pair;$

post: $\exists e[e \in T \wedge Index(e) = I \wedge AccessTable = MakePair(I, Value(e))]$

$\vee \neg \exists e[e \in T \wedge Index(e) = I \wedge AccessTable = MakePair(I, Undefined)]$

(* IsInTable checks to see if an entry whose index is I is in the table. *)

function $IsInTable(T:OwnedTable; I:Domain):boolean;$

post: $IsInTable = \exists e[e \in T \wedge Index(e) = I]$

(* IsOwner checks to see if P is an owner of the entry whose index is I . We overload the \in operator to mean is an element of a sequence. *)

function IsOwner(T :OwnedTable; I :Domain; P :ProcessId):boolean;

post: IsOwner = $\exists e[e \in T \wedge Index(e) = I \wedge P \in Owners(e)]$

(* IsNewestOwner checks to see if P was the most recently added owner. Both AdmitEntry and AddOwner place new owners at the front of the owners sequence. *)

function IsNewestOwner(T :OwnedTable; I :Domain; P :ProcessId):boolean;

post: IsNewestOwner = $\exists e[e \in T \wedge Index(e) = I \wedge P = first(Owners(e))]$

(* IsSoleOwner checks to see if P is the only owner of the entry whose index is I . *)

function IsSoleOwner(T :OwnedTable; I :Domain; P :ProcessId):boolean;

post: IsSoleOwner = $\exists e[e \in T \wedge Index(e) = I \wedge \langle P \rangle = Owners(e)]$

To fully encapsulate the composite types in the system, we also provide the specification of the simple ADT Pair. The source set of the ADT Pair consists of a 2-tuple whose first component, Index, is of type Domain and whose second component, Value, is of type ExtendedRange. A Pair instance is sent by a client for write operations (always with a Value of type Range), to a client in response to read operations, and exchanged between the two servers for updating instances of OwnedTable.

adt:Pair

domain

source set

Pair = **tuple** (Index: Domain, Value: ExtendedRange);

operations

function MakePair(I :Domain; V :ExtendedRange): Pair;

post: MakePair = (I, V)

function IndexOf(P :Pair): Domain;

post: IndexOf = $Index(P)$

function ValueOf(P :Pair): ExtendedRange;

post: ValueOf = $Value(P)$

The simple primitive type `Domain` is used both for flow labels and to compose the ADT's given previously. The type `ExtendedRange` is an alternatively defined type whose values may be either of type `Range` or of type `Bot`, where the type `Bot` is just a simple enumerated type with only one possible value, `Undefined`. The type `ExtendedRange` is used only for type composition in ADT's `OwnedTable` and `Pair`.

```
type Domain = generic;

type ExtendedRange = Range | Bot;
    Range = generic;
    Bot = (Undefined);
```

Figure 4 provides the state transition diagram for server Alpha. The specification of Beta is exactly the same as Alpha, except the process Ids are swapped. The following discussion is from Alpha's perspective. Alpha is specified using five states: `Initial`, `Read_Priority`, `Read_Wait`, `Write_Priority`, and `Write_Wait`. The `Initial` state serves as a mechanism to initialize Alpha's instance of the ADT `OwnedTable` and is never entered again. In the `Read_Priority` state, the specification gives client read requests priority over write requests. Upon satisfaction of a client's read request, Alpha will enter the `Write_Priority` state. In the `Write_Priority` state, client write requests are given priority over read requests. Upon completion of a client's write request, Alpha will enter the `Read_Priority` state. If Alpha's specification correctly specifies this behavior, Alpha is guaranteed to eventually fulfill any read or write requests.

The wait states are entered when a read or write request cannot be immediately satisfied. If Alpha is not an owner of an entry that is in Alpha's `OwnedTable` instance (i.e., the entry is not current) on which a read is requested, Alpha sends a need request (on flow `Need`) to Beta and enters the `Read_Wait` state. When Beta responds with a current value for the entry (on flow `Cur`), Alpha updates its `OwnedTable` instance, completes the read request, and enters the `Write_Priority` state.

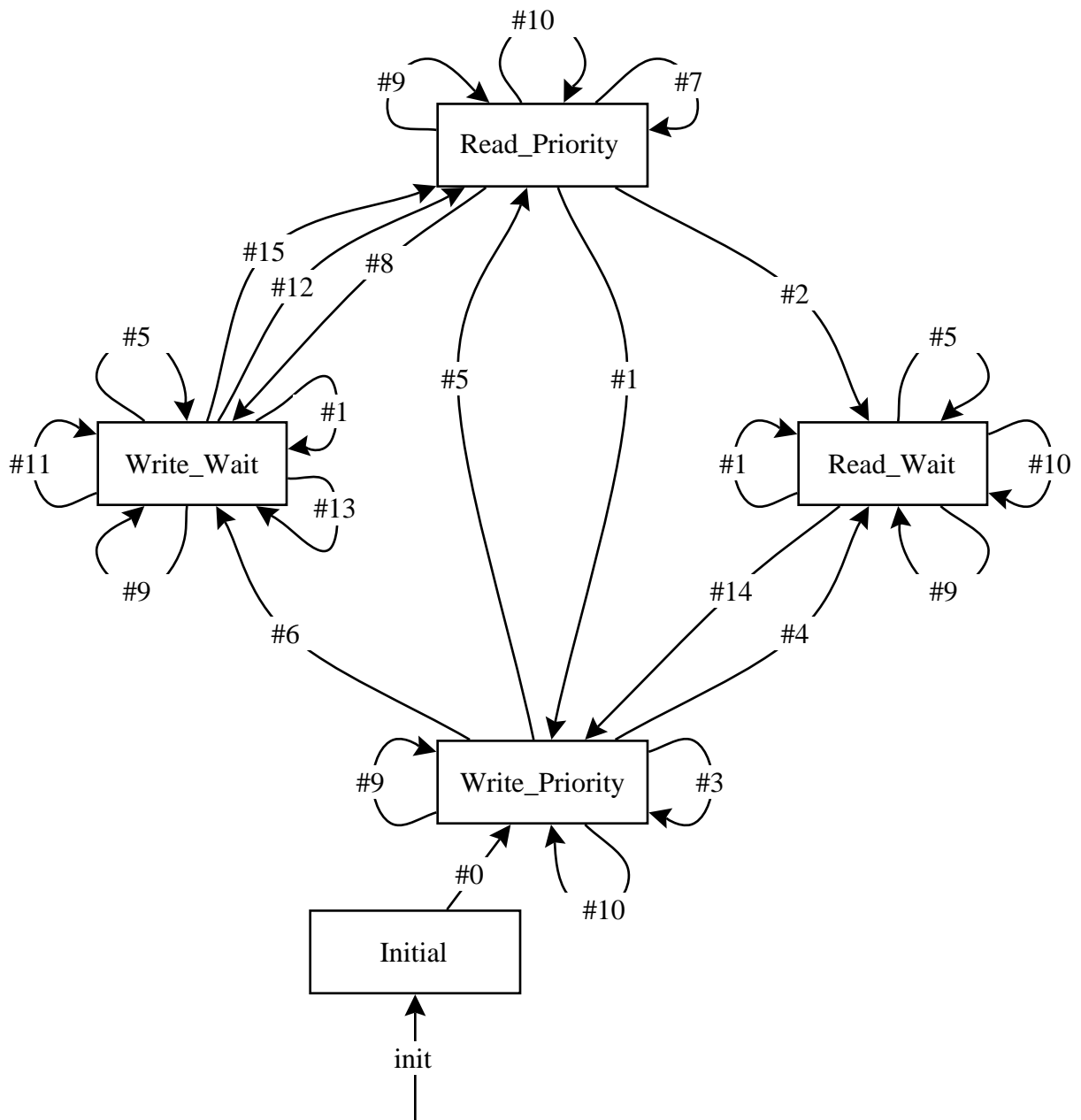


Figure 4: State Transition Diagram for Server Alpha

If Alpha is not the sole owner of an entry on which a write is requested, then either the entry does not exist or Beta is the owner. In either case, Alpha sends a lock request (on flow LockReq) to Beta and enters the Write_Wait state. When Beta receives the lock request, if the entry is not in Beta's OwnedTable instance, Beta adds an entry with an Undefined value and owner Alpha to its OwnedTable instance and sends a lock approval (on flow LockApprv). If the entry is in Beta's OwnedTable instance, Beta makes Alpha the sole owner in its OwnedTable instance and sends a lock approval. In either case, Alpha receives the lock approval, updates its OwnedTable instance completing the write request, and enters the Read_Priority state. However, it is possible that Beta will issue a lock request on the same entry before it receives Alpha's lock request. This race condition is resolved in the specification by implicitly granting approval to the newest owner. The other server is specified to reissue a lock request.⁶ Except for the time period in which a lock approval is in transit from one server to another, the cardinality and set of indices of each server's OwnedTable instance are equal. During the time period in which a lock approval is in transit, the server attempting to write may have one less entry, i.e., the entry it is attempting to write, when that entry has an index that has not yet been added to the table..

The given specification requires that each server give higher priority to servicing requests from the other server, rather than servicing client requests. However, once a client's request has been put into a wait state, getting out of that wait state has priority over requests from the other server.

In the transition labelings we use a prime notation (') on self looping flows to distinguish between the current and new value. The new value is primed. The zeroth transition initializes Alpha's OwnedTable instance and is never used again. Transitions #1 through #9 are initiated by a client. Transitions #10 through #16 are initiated by or responses to the Beta server. The transition labelings follow:

#0 true \models T=NewOwnedTable

Alpha can service a read if it is either an owner or if the given index is not in the OwnedTable

⁶Upon further analysis, we have discovered a deadlock situation which is discussed and resolved later in the paper.

(#1 and #3). However, in the Write_Priority state, no write request can have been received (#3), hence the Entry^- which gives priority to write requests. If Alpha is not an owner and the given index is in the OwnedTable, Alpha will send a need request to Beta and enter the Read_Wait state (#2 and #4), however, once again, in the Write_Priority state, no write request can have been received (#4).

$$\begin{aligned} \#1 \quad & \text{Index}^+ \wedge \text{LockReq}^- \wedge \text{Need}^- \wedge \text{LockApprv}^- \wedge \text{Cur}^- \\ & \wedge (\text{IsOwner}(\text{T}, \text{Index}, \text{Alpha}) \vee \neg \text{IsInTable}(\text{T}, \text{Index})) \\ & \models \text{Val} = \text{AccessTable}(\text{T}, \text{Index}) \end{aligned}$$

$$\begin{aligned} \#2 \quad & \text{Index}^+ \wedge \text{LockReq}^- \wedge \text{Need}^- \wedge \neg \text{IsOwner}(\text{T}, \text{Index}, \text{Alpha}) \wedge \text{IsInTable}(\text{T}, \text{Index}) \\ & \models \text{Need} = \text{Index} \end{aligned}$$

$$\begin{aligned} \#3 \quad & \text{Index}^+ \wedge \text{Entry}^- \wedge \text{LockReq}^- \wedge \text{Need}^- \wedge (\text{IsOwner}(\text{T}, \text{Index}, \text{Alpha}) \vee \neg \text{IsInTable}(\text{T}, \text{Index})) \\ & \models \text{Val} = \text{AccessTable}(\text{T}, \text{Index}) \end{aligned}$$

$$\begin{aligned} \#4 \quad & \text{Index}^+ \wedge \text{Entry}^- \wedge \text{LockReq}^- \wedge \text{Need}^- \wedge \neg \text{IsOwner}(\text{T}, \text{Index}, \text{Alpha}) \wedge \text{IsInTable}(\text{T}, \text{Index}) \\ & \models \text{Need} = \text{Index} \end{aligned}$$

Alpha can service a write if it is the sole owner (#5 and #7), unless a read request in the Read_Priority state overrides the write request (#7), hence the Index^- . If Alpha does not have sole ownership, it will send a lock request to Beta and enter the Write_Wait state (#6 and #8), unless a read request in the Read_Priority state overrides the write request (#8).

$$\begin{aligned} \#5 \quad & \text{Entry}^+ \wedge \text{LockReq}^- \wedge \text{Need}^- \wedge \text{LockApprv}^- \wedge \text{Cur}^- \wedge \text{IsSoleOwner}(\text{T}, \text{IndexOf}(\text{Entry}), \text{Alpha}) \\ & \models \text{T}' = \text{AdmitEntry}(\text{T}, \text{IndexOf}(\text{Entry}), \text{ValueOf}(\text{Entry}), \text{Alpha}) \end{aligned}$$

$$\begin{aligned} \#6 \quad & \text{Entry}^+ \wedge \text{LockReq}^- \wedge \text{Need}^- \wedge \neg \text{IsSoleOwner}(\text{T}, \text{IndexOf}(\text{Entry}), \text{Alpha}) \\ & \models \text{LockReq} = \text{IndexOf}(\text{Entry}) \wedge \text{SaveE}' = \text{Entry} \end{aligned}$$

$$\begin{aligned} \#7 \quad & \text{Entry}^+ \wedge \text{Index}^- \wedge \text{LockReq}^- \wedge \text{Need}^- \wedge \text{IsSoleOwner}(\text{T}, \text{IndexOf}(\text{Entry}), \text{Alpha}) \\ & \models \text{T}' = \text{AdmitEntry}(\text{T}, \text{IndexOf}(\text{Entry}), \text{ValueOf}(\text{Entry}), \text{Alpha}) \end{aligned}$$

$$\begin{aligned} \#8 \quad & \text{Entry}^+ \wedge \text{Index}^- \wedge \text{LockReq}^- \wedge \text{Need}^- \wedge \neg \text{IsSoleOwner}(\text{T}, \text{IndexOf}(\text{Entry}), \text{Alpha}) \\ & \models \text{LockReq} = \text{IndexOf}(\text{Entry}) \wedge \text{SavE}' = \text{Entry} \end{aligned}$$

Alpha will service a “need” from Beta for a table entry in any state, except when Alpha receives a response for a need or lock request it sent to Beta. When Alpha does send a current pair response to Beta, Alpha adds Beta to the owners list of the entry to reflect that Beta has a current entry.

$$\begin{aligned} \#9 \quad & \text{Need}^+ \wedge \text{Cur}^- \wedge \text{LockReq}^- \\ & \models \text{Cur} = \text{AccessTable}(\text{T}, \text{Need}) \\ & \wedge (\text{IsInTable}(\text{T}, \text{Need}) \Rightarrow \text{T}' = \text{AddOwner}(\text{T}, \text{Need}, \text{Beta})) \\ & \wedge (\neg \text{IsInTable}(\text{T}, \text{Need}) \Rightarrow \text{T}' = \text{AddOwner}(\text{AdmitEntry}(\text{T}, \text{Need}, \text{Undefined}, \text{Alpha}), \text{Need}, \text{Beta})) \end{aligned}$$

Alpha will service a lock request on an entry from any state by removing itself from the owners list of that entry in its OwnedTable instance and sending a lock approval (#10 and #11). However, Alpha will not grant approval if it is also requesting a lock on the same index and is the “newer” owner. If both servers are requesting a lock on the same entry, then the server most recently added to the owners list (i.e., the newer owner) of that entry will consider the other server’s lock request to be an implicit lock approval, removing the other server from the owners list and updating its OwnedTable instance (#12). The other server will recognize it is not the newer owner, remove itself from the owners list in its OwnedTable instance if the entry exists and will reissue a lock request (#13).

$$\begin{aligned} \#10 \quad & \text{LockReq}^+ \wedge \text{Cur}^- \\ & \models (\neg \text{IsInTable}(\text{T}, \text{LockReq}) \Rightarrow \text{T}' = \text{AdmitEntry}(\text{T}, \text{LockReq}, \text{Undefined}, \text{Beta})) \\ & \wedge (\text{IsInTable}(\text{T}, \text{LockReq}) \Rightarrow \text{T}' = \text{RemoveOwner}(\text{AddOwner}(\text{T}, \text{LockReq}, \text{Beta}), \text{LockReq}, \text{Alpha})) \\ & \wedge \text{LockApprv} = \text{LockReq} \end{aligned}$$

$$\begin{aligned} \#11 \quad & \text{LockReq}^+ \wedge \text{LockApprv}^- \wedge \text{IndexOf}(\text{SavE}) \neq \text{LockReq} \\ & \models (\neg \text{IsInTable}(\text{T}, \text{LockReq}) \Rightarrow \text{T}' = \text{AdmitEntry}(\text{T}, \text{LockReq}, \text{Undefined}, \text{Beta})) \end{aligned}$$

$$\begin{aligned} & \wedge (\text{IsInTable}(T, \text{LockReq}) \Rightarrow T' = \text{RemoveOwner}(\text{AddOwner}(T, \text{LockReq}, \text{Beta}), \text{LockReq}, \text{Alpha})) \\ & \wedge \text{LockApprv} = \text{LockReq} \end{aligned}$$

$$\begin{aligned} \#12 \quad & \text{LockReq}^+ \wedge \text{IndexOf}(\text{SavE}) = \text{LockReq} \wedge \text{IsNewestOwner}(T, \text{LockReq}, \text{Alpha}) \\ & \models T' = \text{RemoveOwner}(\text{AdmitValue}(T, \text{LockReq}, \text{ValueOf}(\text{SavE}), \text{Alpha}), \text{LockReq}, \text{Beta}) \end{aligned}$$

$$\begin{aligned} \#13 \quad & \text{LockReq}^+ \wedge \text{IndexOf}(\text{SavE}) = \text{LockReq} \wedge \neg \text{IsNewestOwner}(T, \text{LockReq}, \text{Alpha}) \\ & \models (\text{IsInTable}(T, \text{LockReq}) \Rightarrow T' = \text{RemoveOwner}(T, \text{LockReq}, \text{Alpha})) \\ & \wedge \text{LockReq} = \text{IndexOf}(\text{SavE}) \end{aligned}$$

Alpha will complete a read request when a current pair is received. Since the entry may not exist in Alpha's OwnedTable, it is added with Beta as an owner and then Alpha is added as the newest owner.

$$\begin{aligned} \#14 \quad & \text{Cur}^+ \\ & \models \text{Val} = \text{Cur} \wedge T' = \text{AddOwner}(\text{AdmitEntry}(T, \text{IndexOf}(\text{Cur}), \text{ValueOf}(\text{Cur}), \text{Beta}), \text{Alpha}) \end{aligned}$$

Alpha will complete a write request when a lock approval is received.

$$\begin{aligned} \#15 \quad & \text{LockApprv}^+ \\ & \models T' = \text{RemoveOwner}(\text{AdmitValue}(T, \text{LockApprv}, \text{ValueOf}(\text{SavE}), \text{Alpha}), \text{LockApprv}, \text{Beta}) \end{aligned}$$

5.1 Execution Trace

What follows is an example execution of the replicated server specification. In each *step* of the trace given below we provide the following information:

1. Step: A description of the next step in terms of the process that is changing its meta-state.
2. Flow State Changes: A description of the changes to flow values as a result of the step. We list the (complete) flow label, prefix:Type. If the flow is persistent, we list the “now current”

value on the flow. If the flow is consumable, we list the sequence of values of type Type that now appears on the flow. (We denote values of type sequence in angle braces, $\langle \rangle$.)

3. Process States: Each process is listed by name. If a process is idle, we list its current state, its meta-state, and the set of enabled transitions. If the set of enabled transitions is empty, the process is not enabled. If a process is working, we list the resulting state it will be in when it completes its work, its meta-state, and the transition it is working on.

Note that since the terminator “processes” have only one Mealy-machine state (which we have not bothered to name), we just list the current Mealy-machine state of a terminator as “—”. At the “zero-th” step, we provide a complete description of all current flow values. We do not begin with system initialization, but rather jump into the execution trace at an arbitrary point in time.

0. Flow States:
 - T:OwnedTable (into Alpha) =
 - { (1, “one”, \langle Alpha, Beta \rangle), (2, “two”, \langle Alpha \rangle), (3, “five”, \langle Beta \rangle) }
 - T:OwnedTable (into Beta) =
 - { (1, “one”, \langle Alpha, Beta \rangle), (2, “four”, \langle Alpha \rangle), (3, “three”, \langle Beta \rangle) }
 - (and all other persistent flows are undefined and consumable flows are empty)
- Process States:
 - Alpha’s_Clients, —, idle, { #1, #3 }
 - Beta’s_Clients, —, idle, { #1, #3 }
 - Alpha, Read_Priority, idle, { }
 - Beta, Write_Priority, idle, { }
1. Step: Alpha’s_Clients fires transition #1
 - Flow State Changes: (none)
 - Process States:
 - Alpha’s_Clients, —, working on #1
 - Beta’s_Clients, —, idle, { #1, #3 }
 - Alpha, Read_Priority, idle, { }
 - Beta, Write_Priority, idle, { }
2. Step: Beta’s_Clients fires transition #1
 - Flow State Changes: (none)
 - Process States:
 - Alpha’s_Clients, —, working on #1
 - Beta’s_Clients, —, working on #1
 - Alpha, Read_Priority, idle, { }
 - Beta, Write_Priority, idle, { }

3. Step: Beta's_Clients completes transition #1

Flow State Changes:

Index:Domain (into Beta) = <2>

Process States:

Alpha's_Clients, —, working on #1

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Read_Priority, idle, { }

Beta, Write_Priority, idle, { #4 }

Beta's_Clients has issued a ReadEntry on the ADT Table. (Recall that Table is the only perspective the client requires.) Note that Beta is not an owner of the entry with index 2.

4. Step: Alpha's_Clients completes transition #1

Flow State Changes:

Index:Domain (into Alpha) = <2>

Process States:

Alpha's_Clients, —, idle, { #1,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Read_Priority, idle, { #1 }

Beta, Write_Priority, idle, { #4 }

Alpha's_Clients has issued a ReadEntry on the ADT Table. Alpha is an owner of the entry with index 2.

5. Step: Beta fires transition #4

Flow State Changes:

Index:Domain (into Beta) = <> — no values on the flow

Process States:

Alpha's_Clients, —, idle, { #1,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Read_Priority, idle, { #1 }

Beta, Read_Wait, working on #4

Beta gets started on servicing the request to read an entry with index 2. Beta needs a copy of the entry from Alpha.

6. Step: Alpha fires transition #1

Flow State Changes:

Index:Domain (into Alpha) = <>

Process States:

Alpha's_Clients, —, idle, { #1,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Write_Priority, working on #1

Beta, Read_Wait, working on #4

Alpha can satisfy this request directly, since it is an owner.

7. Step: Beta completes transition #4

Flow State Changes:

Need:Domain (into Alpha) = <2>

Process States:

Alpha's_Clients, —, idle, { #1,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Write_Priority, working on #1

Beta, Read_Wait, idle, { }

Beta is waiting for a current version of the entry with index 2 from Alpha. Basically, this state just precludes Beta from servicing another read entry request for an entry Beta does not own.

8. Step: Alpha completes transition #1

Flow State Changes:

Val:Pair (out of Alpha) = <(2, "two")>

Process States:

Alpha's_Clients, —, idle, { #1,#2,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Write_Priority, idle, { #9 }

Beta, Read_Wait, idle, { }

Alpha satisfies its client's read entry request by emitting the pair (2 "two"). Alpha is immediately enabled by the need request from Beta. Alpha's_Clients is now enabled for transition #2 because of the arrival of the pair from Alpha.

9. Step: Alpha's_Clients fires transition #3

Flow State Changes: (none)

Process States:

Alpha's_Clients, —, working on #3

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Write_Priority, idle, { #9 }

Beta, Read_Wait, idle, { }

Alpha's_Clients can fire on any of the three transitions. "Non-deterministically", we chose to trace the execution in which Alpha's_Clients will next generate a write entry request.

10. Step: Alpha's_Clients completes transition #3

Flow State Changes:

Entry:Pair (into Alpha) = <(2, "six")>

Process States:

Alpha's_Clients, —, idle, { #1,#2,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Write_Priority, idle, { #9 }

Beta, Read_Wait, idle, { }

Alpha's_Clients generates a write entry request. Note the specification precludes Alpha from paying attention to this new write request—it must first take care of Beta's request for a current copy of the entry with index 2.

11. Step: Alpha fires transition #9

Flow State Changes:

Need:Pair (into Alpha) = < >

Process States:

Alpha's_Clients, —, idle, { #1,#2,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Write_Priority, working on #9

Beta, Read_Wait, idle, { }

Alpha will pay attention to Beta's need for a current copy of the entry with index 2.

12. Step: Beta's_Clients fires transition #3

Flow State Changes: (none)

Process States:

Alpha's_Clients, —, idle, { #1,#2,#3 }

Beta's_Clients, —, working on #3

Alpha, Write_Priority, working on #9

Beta, Read_Wait, idle, { }

But in the mean time, Beta's_Clients fires to generate a write entry request.

13. Step: Beta's_Clients completes transition #3

Flow State Changes:

Entry:Pair (into Beta) = <(2, "eight")>

Process States:

Alpha's_Clients, —, idle, { #1,#2,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Write_Priority, working on #9

Beta, Read_Wait, idle, { }

Beta's_Clients generates a write request. Note that this write request is for the entry with index 2, which is the same index of the write request pending for Alpha.

14. Step: Alpha completes transition #9

Flow State Changes:

Cur:Pair (into Beta) = <(2, "two")>

T:OwnedTable (into Alpha) =

{ (1, "one", <Alpha, Beta>), (2, "two", <Beta, Alpha>), (3, "five", <Beta>) }

Process States:

Alpha's_Clients, —, idle, { #1,#2,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Write_Priority, idle, { #6 }

Beta, Read_Wait, idle, { #14 }

Alpha sends a copy of the entry with index 2 back to Beta, and notes that Beta is "the more recent" owner of this entry in its copy of the owned table. When Beta is in the Read_Wait state and receives a current pair from Alpha, it must pay attention to that pair.

15. Step: Beta fires transition #14

Flow State Changes:

Cur:Pair (into Beta) = < >

Process States:

Alpha's_Clients, —, idle, { #1,#2,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Write_Priority, idle, { #6 }

Beta, Write_Priority, working on #14

Beta will finally get around to satisfying the read request for the entry with index 2.

16. Step: Beta completes transition #14

Flow State Changes:

Val:Pair (out of Beta) = <(2,“two”) >

T:OwnedTable (into Beta) =

{ (1,“one”,<Alpha,Beta>), (2,“two”,<Beta,Alpha>), (3,“three”,<Beta>) }

Process States:

Alpha's_Clients, —, idle, { #1,#2,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Write_Priority, idle, { #6 }

Beta, Write_Priority, idle, { #6 }

In satisfying its original read entry request, Beta notes that it is the “more recent” owner of the entry with index 2. We will now work with the pending write requests.

17. Step: Beta fires transition #6

Flow State Changes:

Entry:Pair (into Beta) = < >

Process States:

Alpha's_Clients, —, idle, { #1,#2,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Write_Priority, idle, { #6 }

Beta, Write_Wait, working on #6

18. Step: Alpha fires transition #6

Flow State Changes:

Entry:Pair (into Alpha) = < >

Process States:

Alpha's_Clients, —, idle, { #1,#2,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Write_Wait, working on #6

Beta, Write_Wait, working on #6

19. Step: Alpha completes transition #6

Flow State Changes:

SavE:Pair (into Alpha) = (2,“six”)

LockReq:Domain (into Beta) = <2 >

Process States:
 Alpha's_Clients, —, idle, { #1,#2,#3 }
 Beta's_Clients, —, idle, { #1,#3 }
 Alpha, Write_Wait, idle, { }
 Beta, Write_Wait, working on #6

Alpha finds it is not the sole owner of the entry with index 2 and sends a lock request to Beta.

20. Step: Beta completes transition #6

Flow State Changes:
 SavE:Pair (into Beta) = (2, "eight")
 LockReq:Domain (into Alpha) = <2>

Process States:
 Alpha's_Clients, —, idle, { #1,#2,#3 }
 Beta's_Clients, —, idle, { #1,#3 }
 Alpha, Write_Wait, idle, { #13 }
 Beta, Write_Wait, idle, { #12 }

Beta finds it is not the sole owner of the entry with index 2 and sends a lock request to Alpha.

21. Step: Beta fires transition #12

Flow State Changes:
 LockReq:Domain (into Beta) = <>

Process States:
 Alpha's_Clients, —, idle, { #1,#2,#3 }
 Beta's_Clients, —, idle, { #1,#3 }
 Alpha, Write_Wait, idle, { #13 }
 Beta, Read_Priority, working on #12

Beta receives Alpha's lock request for the same entry it is requesting a lock request. Since Beta is the more recent owner, Beta takes Alpha's lock request as an implicit lock approval.

22. Step: Beta completes transition #12

Flow State Changes:
 T:OwnedTable (into Beta) =
 { (1, "one", <Alpha,Beta>), (2, "eight", <Beta>), (3, "three", <Beta>) }

Process States:
 Alpha's_Clients, —, idle, { #1,#2,#3 }
 Beta's_Clients, —, idle, { #1,#3 }
 Alpha, Write_Wait, idle, { #13 }
 Beta, Read_Priority, idle, { }

23. Step: Alpha fires transition #13

Flow State Changes:
 LockReq:Domain (into Alpha) = <>

Process States:
 Alpha's_Clients, —, idle, { #1,#2,#3 }
 Beta's_Clients, —, idle, { #1,#3 }
 Alpha, Write_Wait, working on #13
 Beta, Read_Priority, idle, { }

Alpha receives Beta's lock request for the same entry it is requesting a lock request. Since Beta is the more recent owner, Alpha must reissue a lock request.

24. Step: Alpha completes transition #13

Flow State Changes:

LockReq:Domain (into Beta) = <2>

T:OwnedTable (into Alpha) =

{ (1, "one", <Alpha, Beta>), (2, "two", <Beta>), (3, "five", <Beta>) }

Process States:

Alpha's_Clients, —, idle, { #1,#2,#3 }
 Beta's_Clients, —, idle, { #1,#3 }
 Alpha, Write_Wait, idle, { }
 Beta, Read_Priority, idle, { #10 }

Alpha updates its copy of the owned table to reflect that its entry with index 2 is not current, i.e., Beta is the sole owner, and reissues a lock request.

25. Step: Beta fires transition #10

Flow State Changes:

LockReq:Domain (into Beta) = <>

Process States:

Alpha's_Clients, —, idle, { #1,#2,#3 }
 Beta's_Clients, —, idle, { #1,#3 }
 Alpha, Write_Wait, idle, { }
 Beta, Read_Priority, working on #10

Beta receives Alpha's lock request for index 2.

26. Step: Beta completes transition #10

Flow State Changes:

LockApprv:Domain (into Alpha) = <2>

T:OwnedTable (into Beta) =

{ (1, "one", <Alpha, Beta>), (2, "eight", <Alpha>), (3, "three", <Beta>) }

Process States:

Alpha's_Clients, —, idle, { #1,#2,#3 }
 Beta's_Clients, —, idle, { #1,#3 }
 Alpha, Write_Wait, idle, { #15 }
 Beta, Read_Priority, idle, { }

Beta updates its copy of the owned table to reflect that its entry with index 2 is out-of-date and sends a lock approval to Alpha for index 2.

27. Step: Alpha fires transition #15

Flow State Changes:

LockApprv:Domain (into Alpha) = < >

Process States:

Alpha's_Clients, —, idle, { #1,#2,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Read_Priority, working on #15

Beta, Read_Priority, idle, { }

Alpha receives the lock approval on index 2.

28. Step: Alpha completes transition #15

Flow State Changes:

T:OwnedTable (into Alpha) =

{ (1, "one", <Alpha, Beta>), (2, "six", <Alpha>), (3, "five", <Beta>) }

Process States:

Alpha's_Clients, —, idle, { #1,#2,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Read_Priority, idle, { }

Beta, Read_Priority, idle, { }

Alpha completes the write request on the entry with index 2.

5.2 Deadlock Analysis

The ability to analyze specifications for errors early in the product lifecycle is facilitated by more formal specification techniques. To bear witness to this fact, we discovered an error in the specification while reviewing a draft of this paper. The particular error is a case of deadlock.

The error occurs in the race condition created by a mutual desire to obtain a lock approval on an entry that does not exist in the table. The following trace illustrates the problem:

0. Flow States:

T:OwnedTable (into Alpha) = { }

T:OwnedTable (into Beta) = { }

(and all other persistent flows are undefined and consumable flows are empty)

Process States:

Alpha's_Clients, —, idle, { #1,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Write_Priority, idle, { }

Beta, Write_Priority, idle, { }

1. Step: Alpha's_Clients fires transition #3

Flow State Changes: (none)

- Process States:
 Alpha's_Clients, —, working on #3
 Beta's_Clients, —, idle, { #1,#3 }
 Alpha, Write_Priority, idle, { }
 Beta, Write_Priority, idle, { }
2. Step: Beta's_Clients fires transition #3
 Flow State Changes: (none)
 Process States:
 Alpha's_Clients, —, working on #3
 Beta's_Clients, —, working on #3
 Alpha, Write_Priority, idle, { }
 Beta, Write_Priority, idle, { }
 3. Step: Alpha's_Clients completes transition #3
 Flow State Changes:
 Entry:Pair (into Alpha) = <(1,“won”)>
 Process States:
 Alpha's_Clients, —, idle, { #1,#3 }
 Beta's_Clients, —, working on #3
 Alpha, Write_Priority, idle, { #6 }
 Beta, Write_Priority, idle, { }
 Alpha has issued a WriteEntry on index 1.
 4. Step: Beta's_Clients completes transition #3
 Flow State Changes:
 Entry:Pair (into Beta) = <(1,“one”)>
 Process States:
 Alpha's_Clients, —, idle, { #1,#3 }
 Beta's_Clients, —, idle, { #1,#3 }
 Alpha, Write_Priority, idle, { #6 }
 Beta, Write_Priority, idle, { #6 }
 Beta has issued a WriteEntry on index 1.
 5. Step: Beta fires transition #6
 Flow State Changes:
 Entry:Pair (into Beta) = < >
 Process States:
 Alpha's_Clients, —, idle, { #1,#3 }
 Beta's_Clients, —, idle, { #1,#3 }
 Alpha, Write_Priority, idle, { #6 }
 Beta, Write_Wait, working on #6
 Beta doesn't have sole ownership of an entry with index 1.
 6. Step: Alpha fires transition #6

Flow State Changes:

Entry:Pair (into Alpha) = $\langle \rangle$

Process States:

Alpha's_Clients, —, idle, { #1,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Write_Wait, working on #6

Beta, Write_Wait, working on #6

Alpha also doesn't have sole ownership of an entry with index 1.

7. Step: Alpha completes transition #6

Flow State Changes:

Save:Pair (into Alpha) = (1, "won")

LockReq:Domain (into Beta) = $\langle 1 \rangle$

Process States:

Alpha's_Clients, —, idle, { #1,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Write_Wait, idle, { }

Beta, Write_Wait, working on #6

Alpha finds that it is not the sole owner of an entry with index 1 and issues a lock request. In fact, there is no entry with index 1 in Alpha's table.

8. Step: Beta completes transition #6

Flow State Changes:

Save:Pair (into Beta) = (1, "one")

LockReq:Domain (into Alpha) = $\langle 1 \rangle$

Process States:

Alpha's_Clients, —, idle, { #1,#3 }

Beta's_Clients, —, idle, { #1,#3 }

Alpha, Write_Wait, idle, { #13 }

Beta, Write_Wait, idle, { #13 }

Beta finds that it is not the sole owner of an entry with index 1 and issues a lock request. There is also no entry for index 1 in Beta's table.

At this point each server will fire transition #13, which reissues the lock request on index 1.

This in-turn enables both servers with transition #13 and so on.

To resolve the deadlock, one of the servers must be given preference over the other for writes to new entries. We shall arbitrarily choose Alpha. The unfairness of this preference is minor and, for any particular entry, will not occur again since the entry will now be in the table. The solution is provided by replacing transitions #12 and #13 in server Alpha with the following transition labelings:⁷

⁷The transitions for Beta remain unchanged.

#12 $\text{LockReq}^+ \wedge \text{IndexOf}(\text{SavE}) = \text{LockReq}$
 $\wedge (\text{IsNewestOwner}(\text{T}, \text{LockReq}, \text{Alpha}) \vee \neg \text{IsInTable}(\text{T}, \text{LockReq}))$
 $\models \text{T}' = \text{RemoveOwner}(\text{AdmitValue}(\text{T}, \text{LockReq}, \text{ValueOf}(\text{SavE}), \text{Alpha}), \text{LockReq}, \text{Beta})$

#13 $\text{LockReq}^+ \wedge \text{IndexOf}(\text{SavE}) = \text{LockReq}$
 $\wedge \neg \text{IsNewestOwner}(\text{T}, \text{LockReq}, \text{Alpha}) \wedge \text{IsInTable}(\text{T}, \text{LockReq})$
 $\models \text{T}' = \text{RemoveOwner}(\text{T}, \text{LockReq}, \text{Alpha}) \wedge \text{LockReq} = \text{IndexOf}(\text{SavE})$

In the preceding trace step 8, since the entry is not in the table Alpha will fire with transition #12 (not #13) and complete the write request. Beta will reissue the lock request and Alpha will eventually grant lock approval so Beta can complete it's write request.

6 Conclusions

It has been over 13 years since Boehm first documented that fixing errors during the specification phase of software development is orders of magnitude cheaper than fixing these errors in the maintenance phase [4]. And in the interim, considerable progress has been made in more formal and practical specification techniques for traditional, sequential software systems.

However, more and more systems are not simply sequential. Every application that uses a network-based licensing scheme must deal with this evolving world of distributed and concurrent systems. So just about when we seem poised to apply “mature” formal methods to production environments, the nature of the systems we want to specify is fundamentally changed.

We hope that this paper brings good news relative to this dilemma:

- Specification techniques from Structured Analysis (SA), Data Flow Diagrams (DFD's) in particular, can be applied to the specification of distributed or concurrent systems (in a more straight-forward manner than has previously been proposed).
- These evolving SA specification techniques can be just as “formal” as the many model-based approaches to specification of sequential systems that are gaining in popularity.

- The formalization of SA specification techniques is achieved quite nicely by synthesizing object-oriented, model-based approaches to specification with traditional SA modeling tools: DFD's, DD's and P-Specs.

The examples developed in this paper illustrate a common point of misunderstanding between advocates of model-based specifications and those condemned to implementing modern distributed or concurrent systems. The ADT Table presented in Section 3 does provide one level of specification of the replicated server system specified in Section 5—and it does not require modeling the concurrent aspects of the replicated servers. Unfortunately, this view of the system as ADT Table is only applicable to the client processes, and in many cases some aspects of the concurrent nature of the replicated server systems would have to be included in the client perspective.

Consider adding a function `RemoveEntry` to the ADT Table:

```
function RemoveEntry(T:Table; I:Domain):Table;
  pre:  $\exists e[e \in T \wedge Index(e) = I]$ 
  post:  $RemoveEntry = T - \{e \mid e \in T \wedge Index(e) = I\}$ 
```

This is a likely inclusion in an ADT in a sequential implementation. However, this will not suffice, even for the client perspective, in our replicated server system. The client could use the existing `ReadEntry` function to find out if there is an item in the table with index *I*, as required by the precondition of the additional `RemoveEntry` function. In fact, the client could issue several of these queries for different index values. The “return” to the client of the results of a `ReadEntry` query is an instance of the ADT `Pair`, which contains the index value and a value of type `ExtendedRange`—which will be `Undefined` if the value is not the index of an item in the Table, and the actual value of type `Range` otherwise.

However, the client could receive a return message that a given index value *I* is the index of a particular entry in the table, and, before a subsequent `RemoveEntry` for index *I* is processed, the item might be removed by another client. (In this particular case there is an easy fix. The

precondition to `RemoveEntry` could just be `True`, and then the entire operation could be “atomic” from the perspective of the client. But one can readily imagine cases in which the client would “have to be aware of” the concurrent nature of the “server side” of the system.) The main point here is that there would then have to be some form of specification of the “server side” that does reflect the concurrency. In other words, the simple model-based and sequential view provided by the ADT Table for the replicated server example will not suffice even for understanding the specification for the client processes. (And it certainly will be unsatisfying to the implementers of the replicated server system—a group we probably should not ignore in all this.)

So beware of the falsely comforting claim that all we need are ADT specifications of the type provided by SPECS. But things are not be as dire as they might seem, since we are proposing that a rather familiar specification paradigm, SA, can be elegantly tailored to define concurrent systems at an appropriately abstract and formal level.

Our formalization of DFD’s is based first on using object-oriented, model-based specification techniques to provide better DDE’s. There are three benefits to these more precise DDE’s:

1. These model-based specification techniques necessitate composing a model of the “data” in the system. These models are based on discrete mathematical structures and therefore provide a far more elegant “data model” than we get with implementation-level data structures. (Imagine the added complexity to the specification of the ADT Table from Section 3 if rather than modeling Table as a set of tuples, we work with an implementation data structure like a B-Tree.)
2. Given abstract models of data, operations over these domains can be specified assertively. It is the assertional approach to defining *what* operations do, as opposed to providing an algorithm for *how* the operations work, that allows focus on details that are important at the specification level, without getting embroiled in details that are properly the venue of design and implementation phases.

3. Given that “data on flows” are specified as ADT’s, P-Specs can be expressed using the ADT operations.

The particular language used to provide object-oriented, model-based specifications in DDE’s is not the primary issue here. For example, we could have just as easily used our more recently developed SPECS-C++ language and specified each DDE as a C++ class. We could have also used any of the Larch interface specification languages. [16] is based on the work presented in this paper and contains a formal operational semantics of formalized data flow diagrams that is independent of the particular specification language used in the DDE’s.

Once we have a precise specification of “data”, we can provide a formal definition of the execution of a data flow diagram. With model-based specifications, we have always had a notion of execution, although it is so simple and obvious that we don’t often think of it in these terms. We understand the ADT Table as an abstract interface specification—if the precondition for an operation is satisfied *and the operation is executed*, then the resulting *post state* will be as defined by the postcondition.

The firing of a DFD bubble is loosely analogous to the execution of an ADT operation. If the bubble is idle, in a given state s , and the values on its in-flows satisfy one of the enabling rules on a transition out of s , then the bubble may be selected to fire, in which case the bubble consumes the appropriate in-flows and goes into the working mode. At some time later, the bubble will be selected to fire, will produce out-flow values according to the postcondition on the transition, go into the state “at the end of” the transition and return to the idle mode. Note that this two step firing is needed to appropriately model the concurrent behavior of systems. (See [16] for further discussion.)

However, we have to view the interface behavior of a DFD, i.e., its interface with the terminators, through a sequence of such bubble firings. In fact, the enabling rules (and bubble states) precisely capture the *synchronization primitives* of the concurrent system. We think that this allows for important specification details to be expressed and analyzed early in the development process. The

view of the replicated server system as simply ADT Table is not acceptable, particularly for the implementers of the replicated server system. However, a prototype written in C, and using even a high-level network implementation platform like NCS-RPC [14], will contain too much detail for systematic analysis of issues like synchronization primitives. We think DFD-SPECS provides an appropriate middle ground well-suited to the specification phase of software development.

There are two recent research developments stemming from the work described in this paper that may be worth highlighting. [22] contains a description of an executable formal semantics for a variation of DFD-SPECS. One can think of the operational semantics provided in [22] paper as an interpreter for DFD-SPECS. The operational semantics is written in the programming language Standard ML, and requires input in an ML-like syntax. In this early version of direct execution of formalized data flow diagrams, the specifications contained in DDE's are based on a subset of intrinsic structured types and assertions that are expressions in propositional logic (as opposed to full first order predicate logic expressions). But the semantics of execution built into this interpreter are the same as those described in our paper. This is the first effort to directly execute formalized data flow diagrams.

Since this early effort at direct execution of DFD's, Dr. Baker and his students have focused attention on extending the directly executable subset of object-oriented, model-based specification languages. This effort has focused on the model-based specification language SPECS-C++, which provides features for composing abstract models of C++ class instances and specifying member and friend functions assertionally. Within this SPECS-C++ framework, Wahls has extended direct execution of assertions to include a significant subset of quantified assertions. We refer to this class of assertions as *constructive assertions*. With these results it is possible to take a model-based specification of a C++ class, which can be embedded as special comments directly in the header file for the class, and directly generate a linkable prototype of the class implementation.

Class specifications written in SPECS-C++ can be used as DDE's in DFD-SPECS. Current research and development is focused on this synthesis of SPECS-C++ and DFD-SPECS.

References

- [1] A. L. Baker, J. M. Bieman, and P. N. Clites. Implications for formal specifications: Results of specifying a software engineering tool. In *Proceedings of the Eleventh Annual International Computer Software & Applications Conference (COMPSAC-87)*, Tokyo, Japan, October 1987. IEEE Computer Society and Information Processing Society of Japan.
- [2] Albert L. Baker and David D. Riley. *Data Abstraction and Object-Oriented Software Development*. University Level Computer Science Program, IBM Corporation, 1993. (Course Notes).
- [3] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111, January 1993.
- [4] Barry Boehm. *Software Engineering Economics*. Prentice-Hall, Inc., 1981.
- [5] Yoonsik Cheon and Gary T. Leavens. A quick overview of Larch/C++. Technical Report 93-18, Department of Computer Science, Iowa State University, June 1993. To appear in the *Journal of Object-Oriented Programming*. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.
- [6] David L. Coleman. *Formalized Structured Analysis Specifications*. PhD thesis, Iowa State University, Ames, Iowa 50011, November 1991.
- [7] David L. Coleman and Albert L. Baker. Deliberations on Kung’s process interface modeling. *The Journal of Systems and Software*, 15(2):193–198, May 1991.
- [8] Tom DeMarco. *Structured Analysis and System Specification*. Yourdon, Inc., Englewood Cliffs, New Jersey, 1978.
- [9] Marvin Gore and John Stubble. *Elements of Systems Analysis*. Wm. C. Brown Company, Dubuque, Iowa, 1983.

- [10] D. Hatley and E. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House, New York, 1987.
- [11] Ian Hayes, editor. *Specification Case Studies*. Prentice-Hall, London, 1987.
- [12] John E. Hopcroft and Jeffery D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.
- [13] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International Ltd, London, 1986.
- [14] M. Kong and et. al. *Network Computing System Reference Manual*. Prentice-Hall, Inc., 1990.
- [15] Chenho Kung. Process interface modeling and consistency checking. *The Journal of Systems and Software*, 15(2), May 1991.
- [16] G. T. Leavens, T. Wahls, A. L. Baker, and K. Lyle. An operational semantics of firing rules for structured analysis style data flow diagrams. 1993. (submitted).
- [17] Gary T. Leavens and Yoonsik Cheon. Larch/C++ reference manual. Available by anonymous ftp from ftp.cs.iastate.edu., 1993.
- [18] Gary T. Leavens and Yoonsik Cheon. Preliminary design of Larch/C++. In U. Martin and J. Wing, editors, *Proceedings of the First International Workshop on Larch, July, 1992*, Workshops in Computing, pages 159–184. Springer-Verlag, NY, 1993.
- [19] Tadao Murata. Modeling and analysis of concurrent systems. In C. R. Vick and C. V. Rammamoorthy, editors, *Handbook of Software Engineering*, pages 39–63. Van Nostrand Reinhold Company Inc., New York, 1984.
- [20] James L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1981.

- [21] John Rushby, Friedrich von Henke, and Sam Owre. An introduction to formal specification and verification using EHDM. Technical report, Computer Science Laboratory, SRI International, 1991.
- [22] T. Wahls, A. L. Baker, and Leavens G. T. A formal (and executable) semantics for rt-specs. 1993. (submitted).
- [23] Paul T. Ward. The transformation schema: An extension of the data flow diagram to represent control and timing. *IEEE Transactions on Software Engineering*, SE-12(2):198–210, February 1986.
- [24] Paul T. Ward and Stephen J. Mellor. *Structured Development for Real-Time Systems*, volume 3: Implementation Modeling Techniques. Yourdon, Inc., Englewood Cliffs, New Jersey, 1986.
- [25] Stephen S. Yau and Mehmet U. Cadayan. Distributed software system design representation using modified petri nets. *IEEE Transactions on Software Engineering*, SE-9(6):733–745, November 1983.
- [26] Edward Yourdon. *Modern Structured Analysis*. Yourdon Press computing series. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.



IOWA STATE UNIVERSITY

OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

SCIENCE
with
PRACTICE

Tech Report: TR94-04
Submission Date: March 4, 1994