PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

PIONEERS IN METHYLENE BLUE TESTING SINCE 1974

19000 COUNTY ROAD 3 BURLINGAME MN 55427 USA
TEL 612 436 7007 FAX 612 436 7007 Tlx 5100000400

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Describing Groups of Interacting Objects Using Path Expressions

by
Gregory Adams, B.C.S.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

School of Computer Science
Carleton University
Ottawa, Ontario
March 1992

©1992, Gregory Adams

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-75981-X

Canada

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of this thesis

**Describing Groups of Interacting Objects
Using Path Expressions**

submitted by Gregory Adams, B.C.S.

in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Thesis Supervisor

Thesis Supervisor

Chairman, School of Computer Science, Carleton University

15 May 1992

Date

Abstract

We propose an improved notation (PEGs) for describing the behaviour of groups of interacting objects within an object-oriented system. Interactions are expressed using path expressions. Interaction groups can be composed into abstraction hierarchies or composed to form complex groups. Current methods of describing systems of objects are examined. Frameworks and the application of the notation to frameworks is discussed. We illustrate our notation using Smalltalk/V's Pane/Dispatcher framework.

Keywords: framework, path expression, regular expression, contract, object interaction, object-oriented programming, Smalltalk, object specification, object description

Acknowledgements

I would like to thank the following people:

Dave Thomas

For his support as my supervisor, but most of all, for being a friend and one of the most giving and supportive people I know.

Jean-Pierre Corriveau

For his support as my supervisor and as a friend.

Object Technology International

For allowing me the time and resources to complete my thesis.

Thai Nguyen

For helping out with some of the figures. Thai's always there when you need him the most.

Corey Guntzel

For reading and critizing this document, and for calling me every morning to make sure I was still alive.

Mike Wilson and John Duimovich

For reading and criticizing this document - and for being cool guys.

My Parents

For being who they are.

Contents

1	Introduction	1
1.1	Objects and Classes	3
1.2	Subclasses, Superclasses and Abstract Classes	3
1.3	Groups of Objects and their Description	4
1.4	Summary Of Notations	7
1.5	Motivation	8
1.6	Statement of Intent	10
1.7	Contribution	11
1.8	Thesis Organization	11
2	Descriptive Approaches	12
2.1	Schlaer and Mellor	12
2.1.1	State Model Diagram	13
2.1.2	Object Communication Model Diagram	14
2.1.3	Action Data Flow Diagrams	15
2.1.4	Discussion	17
2.2	Rumbaugh et al.	21
2.2.1	Discussion	24
2.3	Bear et al.	27
2.3.1	Configuration Diagrams	27
2.3.2	State Diagrams	27
2.3.3	Discussion	31
2.4	Wirfs-Brock et al.	32
2.4.1	Collaboration Graph	32
2.4.2	Subsystem Diagram	33
2.4.3	Discussion	35
2.5	Pernici	37
2.5.1	Diagrams	39
2.5.2	Special Rules	40
2.5.3	Discussion	41
2.6	Helm et al.	43
2.6.1	Discussion	47
2.7	Summary	48
3	The PEGs Notation	50
3.1	Path Expressions	52
3.2	Predicate Path Expressions	55
3.3	Procol and Ada Packages	55
3.4	Why Path Expressions?	57
3.5	Pane/Dispatcher/Model Framework	58

3.6	PEGs	60
3.6.1	Path Expressions	61
3.6.2	Groups	62
3.6.3	Inheritance	74
3.6.4	Composition	80
3.6.5	Describing Classes and Methods	84
3.6.6	Equivalences	88
3.7	PEGs Syntax	89
3.7.1	Groups	89
3.7.2	Class	92
3.7.3	Method	92
3.7.4	BNF Summary	93
3.8	Summary	93
4	Conclusion	96
4.1	Future Research	98
4.2	The Final Word	101
5	References	103
A	Appendix A: Pane/Dispatcher Group	107
A.1	PaneDispatcher Group	107
A.2	NewPane Class	109
A.3	NewPane Methods	109
A.4	NewDispatcher Class	118
A.5	NewDispatcher Methods	118
B	Appendix B: Subpane/Dispatcher Group	125
B.1	SubPaneDispatcher Group	125
B.2	NewSubPane Class	125
B.3	NewSubPane Methods	126
C	Appendix C: TopPane/TopDispatcher Group	129
C.1	TopPaneDispatcher Group	129
C.2	NewTopPane Class	131
C.3	NewTopPane Methods	131
C.4	NewTopDispatcher Class	140
C.5	NewTopDispatcher Methods	140
D	Appendix D: Sample Window Group	147
D.1	SampleWindowFramework Group	147
D.2	SampleModel Class	148
D.3	SampleModel Methods	148

List of Figures

1	Summary of notations.	9
2	State model for a train object	13
3	OCM diagram for a microwave oven	16
4	An action data flow diagram	17
5	A thread of control diagram	20
6	Portion of a Schlaer and Mellor trace.	20
7	State diagram for a vending machine	21
8	State diagram with contours	23
9	Event trace diagram	25
10	Event trace.	26
11	Extended state diagram notation	26
12	Configuration diagram for an alarm clock system	28
13	State diagram for an alarm clock object	29
14	Some transitions for an alarm clock object.	30
15	Collaboration graph for an ATM system	34
16	Subsystem diagram for a portion of the ATM system	36
17	Role transitions of a car	39
18	State diagram of a role	40
19	An example contract for a subject and views.	46
20	Path expression operators and their meaning.	53
21	Bruegge and Hibbard's path expression operators and their meaning.	54
22	PEGs' path expression operators and their meaning.	61
23	Section 1 of the Pane/Dispatcher group.	65
24	Section 2 of the Pane/Dispatcher group.	68
25	Section 3 of the Pane/Dispatcher group.	69
26	Section 4 of the Pane/Dispatcher group (part 1 of 2).	72
27	Section 4 of the Pane/Dispatcher group (part 2 of 2).	73
28	The create path.	75
29	A subset of the TopPane/TopDispatcher group.	77
30	A subset of the TopPane/TopDispatcher group (cont. 2).	78
31	A subset of the TopPane/TopDispatcher group (cont. 3).	79
32	A subset of the Sample Window group.	82
33	A subset of the Sample Window group (cont. 2).	82
34	Method specification	87
35	Summary of notations with PEGs.	95

Chapter 1: Background

1 Introduction

An object-oriented system organizes a computation into a collection of objects consisting of structure and behaviour. The system's behaviour is characterized by its objects and their interactions. Objects within the system communicate through the exchange of messages. The result is a cooperative effort to accomplish a set of tasks.

One of the main features of object-oriented systems is their support of software reuse. As Gibbs et al. [Gibbs, et al. 90] and Wirfs-Brock and Johnson. [Wirfs-Brock and Johnson 90a] have observed, reuse typically occurs either through the reuse of groups of objects interacting to accomplish tasks, or through the specialization of frameworks, that is, collections of classes whose behaviour is intended to be specialized. Smalltalk/V's pane/dispatcher/model [Digitalk 88] and Smalltalk-80's Model/View/Controller [Goldberg and Robson 85] are two of the better known examples of this type of framework.

Obtaining satisfactory levels of reuse requires an understanding of a system's objects. This in turn implies the need for a descriptive notation to capture the interactions within an object-oriented system. Unfortunately, capturing the dynamic aspect is one of the main difficulties with describing object-oriented systems [Pernici 89, pp. 90]. [Bear, et al. 90, pp. 28], [Rumbaugh, et al. 91, pp. 84]. Although the object-oriented design literature contains numerous notations for describing the relationships among objects, these representations typically address only the static aspects of the system. Helm et al. [Helm, et al. 90] note that

existing notations fail to adequately capture the dynamic/behavioural interaction aspects of such a system. Moreover, these notations do not support frameworks [Wirfs-Brock and Johnson 90a].

In this thesis, we examine the issues associated with describing groups of interacting objects and frameworks. Current descriptive techniques are examined and a novel descriptive notation called Path Expression Groups (PEGs) is introduced. The PEGs notation allows the identification of groups of interacting objects. Groups utilize the well-known concepts of regular expressions and path expressions to support the description of objects and their interaction. The thesis is motivated by the ideas presented in [Helm, et al. 90], [Holland 92] and [Campbell and Habermann 73].

Within the object-oriented industry there is a requirement for a mechanism to describe the interactions among objects. PEGs is a preliminary investigation into a descriptive notation intended to provide an approximation of the behaviour of an object-oriented system. Our goal is to find a descriptive common ground between the loose informalisms of object-oriented analysis and the rigour of formal specifications. The class specification notations of Thomson [Thomson 92] and Cook [Cook 92] also use a balance between formal and informal techniques. Thomson notes, "while providing a precise model for studying and understanding types, most formal models do not scale well to production systems" and goes on to note that they are, "sometimes harder to understand than the implementations they describe". Similarly, Skuce [Skuce 88] observes, "...a wide spectrum from very formal techniques beyond the comprehension of most programmers and users, to very informal techniques, which use unconstrained natural language and hence are extremely error prone to semantic confusion".

In our investigation we focus on traditional sequential object-oriented systems; real-time and concurrent systems are beyond the scope of this thesis. The notation is not intended to be executable. PEGs by itself does not imply a design method-

ology but can be an activity performed during design. Consideration of the formal semantics of PEGs will be given only after further experiments have been conducted using the notation.

We will now review some the fundamental object-oriented concepts and terminology.

1.1 Objects and Classes

An *object* consists of structure and behaviour (operations). Statically an object is characterized by the set of *services* (operations) it provides to other objects. Communication between objects is carried out through the exchange of messages. A message specifies the operation the sender wants the receiver to perform on receiving the message. Messages do not specify how the receiver should perform the operation. In Smalltalk, a specific implementation of an operation is called a *method*.

A *class* is a template specifying the structure and behaviour of all objects of a specific kind. Objects constructed according to a given template are *instances* of the associated class. For example, all strings are instances of the class String and as such the behaviour and structure of all strings is described within the class String.

1.2 Subclasses, Superclasses and Abstract Classes

In describing object-oriented systems in general, Wegner [Wegner 87] defines a *subclass* as a class whose instances inherit structure and behaviour from another class, called its *superclass*. A subclass can refine operations and add to the structure of instances. Refinement can be either in the form of new behaviour or the replacement or modification of inherited behaviour. Classes are arranged into an inheritance hierarchy based on their subclass-superclass relationships. Multiple inheritance allows a class to have more than one superclass, with the resulting behaviour being mixed in from all superclasses. Rumbaugh [Rumbaugh, et al. 91, pp. 67] observes, "Dealing with lack of multiple inheritance is really an implementation issue...". Hence,

without loss of generality, we focus on a single inheritance system.

Abstract classes do not have instances constructed from themselves. Abstract classes typically provide either a default behaviour for operations or provide an empty operation template. Subclasses are expected to specialize (reimplement) the behaviour provided by an abstract class.

1.3 Groups of Objects and their Description

An object interacts with other objects in complex ways. Consequently, it is insufficient to describe an object as an entity isolated from all other objects. Indeed, as Beck and Cunningham [Beck and Cunningham 89] observe, “...no object is an island. All objects stand in relationship to others on whom they rely for services”. As object-oriented systems proliferate and increase in size, so to does the need for descriptive facilities through which they can be understood. The importance of this is demonstrated by the amount of literature devoted to the topic. Within the object-oriented community, especially the design domain, numerous notations have been proposed for capturing individual object behaviour and the behaviour of an object-oriented system. We will now survey some of these notations to introduce the terminology. The notations will be further examined in Chapter 2.

Schlaer and Mellor [Schlaer and Mellor 92] capture the interactions within an object-oriented system using three diagrams, a state model diagram, an Object Communication Model (OCM) Diagram, and an Action Data Flow Diagram (ADFD). For each object in the system a state model diagram is constructed. These diagrams are similar to traditional state transition diagrams (STDs) and express the view that an object progresses through a “lifecycle – an orderly pattern of behaviour” [ibid.]. An STD contains a set of states the object can be in, and, a collection of directed transition arcs describing the events which cause the object to move from one state to the next. Transition arcs are labelled with the event causing the transition to be taken. A set of actions are associated with each state.

These actions are performed on entering the state and may generate events to other objects in the system.

Schlaer and Mellor capture the interactions between objects using an Object Communication Model diagram. Objects and external entities (eg. a user and a physical device) interact through the exchange of events, modelled by directed arcs between the objects. The third diagram, an Action DFD captures the behaviour of the system from the perspective of the actions it performs. ADFDs are similar to classical data flow diagrams [Wirfs-Brock and Johnson 90a, pp. 108]. Objects participating in an action are modelled by data stores. Thus, interactions are indirectly modelled by the occurrence of data stores.

Rumbaugh et al. [Rumbaugh, et al. 91, pp. 85-96], employ state diagrams to capture the behaviour of an object-oriented system. Their state diagrams incorporate Harel's [Harel 87] contour notation. As with Schlaer and Mellor's approach, states use actions to generate events. Rumbaugh et al. add the ability to place actions, termed "activities" (to denote their typically shorter life span than complex actions), on the transition arcs. States can be arranged into a generalization hierarchy and represented using contours with substates inheriting the transitions of their superstate. Activities and events can be expanded into lower level state transition diagrams. Rumbaugh et al. describe a particular execution of an object-oriented system in terms of the events it exhibits. Events which occur during the execution are captured in an event trace and expressed as an event trace diagram.

Bear et al. [Bear, et al. 90] describe object-oriented systems using a hybrid notation combining the approaches of Rumbaugh et al. and Schlaer and Mellor. An object-oriented system's behaviour is described using a configuration diagram and a collection of state charts. State charts model the behaviour of an individual object using a Harel style notation similar to Rumbaugh et al. Inter-object communication is captured in a configuration diagram similar to Schlaer and Mellor's object communication model diagram. However, a configuration diagram explicitly

identifies all services required and provided by each of the objects. Interactions are described by a connection between a required service and a provided service. State diagrams of Bear et al. support the identification of “observer” events; those events which do not cause a change of state. The state diagram is accompanied by a collection of transition specifications defining a predicate precondition (guard condition), the event (service being requested) and a predicate postcondition. Hidden attributes and invariant expressions can be associated with any state. The behaviour of a particular execution of the system is captured using an event trace similar to Rumbaugh et al.

Wirfs-Brock et al. [Wirfs-Brock, et al. 90b] identify responsibilities, the obligation of one object to provide another object with a given service. Cohesive sets of responsibilities are organized into disjoint sets called contracts. An object is characterized by the contracts it supports. A subclass supports the contracts identified by its superclass. Groups of classes that collaborate to support a set of contracts are formed into a subsystem. Two diagramming notations are used to capture system behaviour, subsystem diagrams and collaboration graphs. Subsystem diagrams represent the subsystem in terms of nested contours with each contour representing an object and the contracts it supports. A collaboration graph describes collaborations between objects. A collaboration is denoted by a directed arrow connecting two objects. Contours denote subclassing.

Pernici [Pernici 89] proposes describing an object in terms of the roles it can play. Roles are state machines containing a collection of attributes local to the role, messages that are sent or received while in the role, a set of states, and a collection of transition rules. Transition rules specify a starting state, the message that can be accepted and optionally, the object that sends it. A transition rule also specifies the next state and any message that should be sent to another object. Interactions are thus described within the transition rules. Separate transitions rules are used to define constraints (eg. preconditions) or messages that are identical in meaning

while in a given role.

Helm et al. [Helm, et al. 90] and Holland [Holland 92] organize an object-oriented system into groups of interacting objects called contracts. A contract defines a set of participating objects and their behavioural dependencies on one another. Inter-object dependencies are expressed in terms of the messages exchanged and the invariants maintained by the objects. Contracts can be arranged into inheritance hierarchies and composed of other contracts. A contract defines each participating object, the invariants maintained by the contract, and a specification of how the contract is instantiated. A contract also specifies any other contracts that are being refined or included by the given contract. For each participating object, a collection of operations are defined. The behaviour of each operation is described in an algebraic style notation supporting message passing, if-then-else and looping. Holland [Holland 92] extends the contract notation of Helm et al. by requiring participants to explicitly declare whether an operation is visible outside of the contract. Contracts are orthogonal to classes and place the emphasis on capturing groups of interacting objects. Contracts can be mapped to an actual implementation using conformance declarations. A conformance declaration declares which classes implement a participant's operations. A participant's operations can potentially be spread across a hierarchy of classes.

1.4 Summary Of Notations

Existing notations generally capture the behaviour of an object-oriented system in a static manner and tend to hide object interactions. Helm et al. introduces the concept of contracts to support groups of interacting objects. The main contribution of Helm et al. is the ability to form abstract hierarchies and compositions of groups of interacting objects. Unfortunately, within a contract, especially complex contracts involving compositions, the patterns of interaction are not explicit and must again be inferred from code.

Figure 1 summarizes the above notations. In Chapter 2 each of the notations will be examined in greater detail.

1.5 Motivation

The complexity of a system is, in part, the essence of the system and it is important to describe the system in such a manner as not to abstract out its complexity [Brooks 87], [Harel 92]. In an object-oriented system this complexity arises from the interaction of the participating objects. Although an object can communicate with any other object, typically only a few patterns of communication are observed [Schlaer and Mellor 92]. It is important to identify these important interactions [Lalonde, et al. 91], [Lalonde 92] that are the essence of the system.

Wirfs-Brock et al. [Wirfs-Brock, et al. 90b, pp. 13] define a framework to be a group of classes specifically designed to be refined and used as a group. Wirfs-Brock et al. describe frameworks as white boxes whose structure and behaviour must be able to be quickly understood in order to assure their reuse. Pane/dispatcher/model and model/view/controller are user interface frameworks in Smalltalk. Choices [Madany, et al. 89], [Russo and Campbell 88], [Russo and Campbell 89], [Zweig and Johnson 90] is an operating system framework. Existing notations do not adequately describe frameworks [Wirfs-Brock and Johnson 90a].

¹Although state diagrams like Harel's can provide a formal description, the above notations do not use them for that purpose. The diagrams are used to capture analysis and design information and are insufficient to provide a formal specification of the system.

	Schlaer and Mellor	Rum- baugh et al.	Bear et al.	Wirfs- Brock et al.	Pernici	Helm et al.
diagram based	yes	yes	yes	yes	no	no
design methodology	yes	yes	no	yes	no	no
groups	yes	no	no	no	no	yes
group inheritance	no	no	no	no	no	yes
group composition	yes	no	no	yes	no	yes
event traces	yes	yes	yes	no	no	no
object description	state diagram	harel diagram	harel diagram	collab- orations	roles	formal operat- ions
static interaction picture	yes	no	yes	yes	no	no
dynamic of interactions	no	no	no	no	no	no
formal ¹ specification	no	no	no	no	no	yes
identifies invariants	no	no	yes	no	yes	yes

Figure 1: Summary of notations.

1.6 Statement of Intent

In general, the above notations focus on the individual object and its internal behavior. The description of object interactions is limited to a static snapshot capturing the system at an instant in time when every interaction simultaneously occurs.

An objective is therefore to describe the interactions of objects. An object-oriented system is characterized by the patterns of interactions it exhibits, and it is through descriptions of these interactions that the essence of such a system can be understood. In addition, the aim is to capture the **typical** patterns of interaction that occur, rather than exhaustively defining all possible interactions that may occur.

An object does not exist in isolation, it cooperates with other objects to accomplish a set of tasks. Furthermore, as Gibbs et al. [Gibbs, et al. 90] and Wirfs-Brock and Johnson [Wirfs-Brock and Johnson 90a] have observed, reuse typically occurs either through the reuse of groups of objects interacting to accomplish tasks, or through the specialization of frameworks. As a consequence, the notation must be capable of representing the system as groups of interacting objects. Groups must be first class objects supporting inheritance and composition. Failure to provide group inheritance and composition will compromise system modularity, increase complexity and potentially reduce reusability. Abstract groups must be supported, and be able to be specifically identified as providing an abstract behaviour. It is through the identification of abstract behaviour, and a description of the behaviour itself that clients will have sufficient understanding to carry out reuse.

Ideally a notation for describing system behaviour should be based on existing terminology and notations. The introduction of completely new concepts can make systems described using the notation difficult to comprehend, forcing users to invest a considerable amount of time and effort understanding the notation. Notations that direct the user's energy away from understanding the system and place it on understanding the notation should be avoided.

Real-time and concurrent systems are beyond the scope of this thesis. A single inheritance environment is assumed. The absence of multiple inheritance can be overcome through implementation strategies [Rumbaugh, et al. 91, pp. 67] and can be omitted without loss of generality.

1.7 Contribution

The PEGs notation presented in this thesis is not intended to be yet another design approach. It is assumed the objects to describe already exist. Furthermore, the PEGs notation is not a formal (provable) specification such as those of [Helm, et al. 90], [Holland 92], [Wills 91] and [Wills 92].

PEGs is based on path expressions, a notation that has existed for some time which in turn is based on the familiar regular expression mechanism. PEGs' path expressions provide an abstract representation of the externally visible behaviour of an object-oriented system as depicted through its event traces. PEGs supports the grouping of interacting classes into groups whose behaviour is defined by path expressions. These path expression groups (PEGs) are themselves objects and can be organized into abstraction hierarchies and combined to form complex groups.

1.8 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 reviews background material and existing notations. Chapter 3 presents the path expression group notation we propose and formally defines its syntax. The final chapter summarizes the notation and its contribution, and examines directions for future research.

Chapter 2: Notations

2 Descriptive Approaches

In this chapter we examine some of the existing notations for describing objects and their interactions. This survey is not intended to be exhaustive, it presents notations typical of those which capture the behaviour of objects or support groups or sub-systems of objects. The notations of [Schlaer and Mellor 92], [Rumbaugh, et al. 91] and [Wirfs-Brock, et al. 90b] are associated with particular object-oriented design techniques. This chapter does **not** assess the usefulness of these notations as applied within their authors' design methodologies. The notations are critiqued in terms of their ability to capture the dynamic aspect of interacting objects and the objectives outlined in Chapter 1.

We will now present some of the existing notations. The description of each notation is followed by a discussion of the notation.

2.1 Schlaer and Mellor

Schlaer and Mellor [Schlaer and Mellor 92] capture the behaviour of an object-oriented system using three diagrams, a state model diagram, an object communication model diagram (OCM diagram), and an action data flow diagram (ADFD). State model diagrams present the behaviour of individual objects. OCM diagrams describe a higher level view showing the objects and the messages they exchange. Action data flow diagrams represent the actions performed by the system.

2.1.1 State Model Diagram

State model diagrams (SMDs) describe the behaviour of a single object. An SMD is constructed for each object in the system. An SMD contains:

- A set of states (squares in figure 2) the object passes through during its existence.
- A collection of transitions (directed arcs in figure 2) specifying when the object moves from one state to the next.
- A set of actions (text appearing below a state in figure 2), one for each state. Actions specify the behaviour of the object once the state is entered.

Figure 2: State model for a train object

Transitions specify an event and the state the object should move to if the event occurs. An *event* is the reception of a message. A unique label is associated with each event. In figure 2, receipt of the event *T2 (at platform)* causes the train object to move from state 2 (approaching station) to state 3 (securing arrival). On entering state 3 the train object performs the actions:

Tell all passenger cars to open doors

Set timer for time to depart

Object interaction occurs when the action code contained in the state of some object generates an event to another object. The syntax for generating an event is:

Generate eventLabel : eventMeaning (eventData)

This specifies the event whose label is *eventLabel*, is generated. The object to receive the event is determined by searching all state model diagrams for an event with the specified label. Event labels are unique for each object. To illustrate, assume an object other than the train object contains the following action code in one of its states:

Generate T2: At platform (train ID)

This action generates event T2 which causes the train object to move from state 2 to state 3 as described above and shown in figure 2.

2.1.2 Object Communication Model Diagram

Object interactions are captured in an object communication model diagram (see figure 3). All objects (rounded boxes in figure 3) and external entities (squares in figure 3) participating in the system are depicted in this diagram. Examples of external entities are: a user and a physical device.

All events exchanged by objects in the system are specified by directed arcs from the generating object to the receiving object. In figure 3 the *Oven* object

generates event *L2 Turn off light (light ID)*. The recipient of event L2 is the *Light* object. OCM diagrams can be composed of other OCM diagrams, thus allowing the system to be broken into subsystems. Schlaer and Mellor remark, "By convention the OCM is laid out with the most knowledgeable and powerful objects near the top of the page, and with the less knowledgeable successively down the page." [Schlaer and Mellor 92, pp. 89].

2.1.3 Action Data Flow Diagrams

An action DFD captures the behaviour of the system from the perspective of the actions it performs. An ADFD is constructed for each state's action code. An ADFD diagram describes the sequence of steps (ellipsis in figure 4) performed by the action. Interactions are indirectly modelled by flow lines going to, or coming from, data stores (parallel horizontal lines in figure 4) representing objects. Also, flow lines without a destination are indirect indications of object interaction. Some non-terminating arcs may also indicate internal transitions, that is, the generation of an event by an object which is subsequently received by the same object. The ADFD in figure 4 describes the actions of state 3 of the train object shown in figure 2. This ADFD indicates the state's action finds a train's passenger cars by processing the id of all passenger cars ({} denotes repetition) and the train's id. The resulting data, the set of car id's for the train's cars, subsequently *flows* into the next step where events are generated to open the doors of those cars. {} denotes repetition of the event.

Figure 3: OCM diagram for a microwave oven

Figure 4: An action data flow diagram

2.1.4 Discussion

State model diagrams obscure the interactions between objects by hiding them in state actions and indirectly specifying the object to whom an event is intended. Generated events do not identify the target object, instead the target must be inferred by determining the object whose state model diagram responds to the event. Specifying only the event is reasonable if more than one object can respond to the event and thus event sharing is possible; Schlaer and Mellor do not permit shared events.

Although state model diagrams can be broken into a set of substate model diagrams, Schlaer and Mellor stop short of decomposing an SMD into levels of abstraction. In other words, there is no concept of an inheritance hierarchy for states (see Rumbaugh et al. below). Hence, the introduction of subdiagrams does little more than form a flat partitioning of the original diagram.

There is insufficient support for an object to be in complex states. For example, assume a car object goes through the states of being made, being insured, and being owned, where the latter two states could exist either independently or simultaneously. Representing this requires an extra state insured/owned and transitions to and from that state. This could potentially result in an explosion of the diagram and an accompanying decrease in readability.

The object communication model diagram helps to clarify the interactions between objects. As mentioned above, state model diagrams obscure object interactions. However OCM diagrams help to alleviate this problem. Although OCM diagrams depict the events exchanged between objects, they do so in a static manner showing the set of all events that could ever occur. Hence, there is no support for the actual patterns of interaction that might occur. Schlaer and Mellor identify the usefulness of a thread of control (see figure 5 and figure 6) to describe the interactions of a particular execution of the system. Schlaer and Mellor limit their use of threads to the post examination of individual executions of the system.

Allowing OCM diagrams to be composed of other OCM diagrams facilitates the decomposition of the system into subsystems. This helps support the principles of modularity and serves as a mechanism for the decomposition of the system in a top down fashion.

Schlaer and Mellor suggest an OCM diagram can be arranged in a tree-like fashion with the “more knowledgeable” objects at the top and with communications flowing in a downward manner towards the “less knowledgeable” objects. This is too optimistic; in describing systems such as the model/view/controller of

Smalltalk-80 [Goldberg and Robson 85], or the ATM bank example of Wirfs-Brock et al. [Wirfs-Brock, et al. 90b, pp. 151], communications may not flow in a simple tree like manner as depicted by Schlaer and Mellor. For example, in figure 3, objects such as the light may generate events when they have operated beyond their expected lifetime. Consequently, when modelling more complex examples, diagrams such as that of figure 3 can become cumbersome.

An ADFD presents what can be best described as a schizophrenic view of objects motivated by the author's prior involvement with data flow diagrams. Specifically, objects think they are objects at times, signified by their absence at the end of an arc, and a bucket of bits, signified by a data store, at other times. The granularity of an ADFD is too fine to be of use in understanding the behaviour of a system². An ADFD is associated with the action of every state of every object in the system. The result is an explosion in the number of diagrams. ADFDs focus on the behaviour of an operation; a level below the behaviour of the object as a whole.

²This does not imply ADFDs are not helpful within the design methodology advocated by Schlaer and Mellor

Figure 5: A thread of control diagram

State Machine	Action	time Component	Duration
Customer 10	Wait for Clerk	action time	0:01
Clerk	Serving	dwell time (remainder)	3:00
Clerk	Done	action time	0:06
Clerk	Done	dwell time	0:00
Clerk	Idle	action time	0:01
Assigner	Wait for Clerk	dwell time	0:01
Assigner	Assigning	action time	0:12
			3:21 (total)

Figure 6: Portion of a Schlaer and Mellor trace.

2.2 Rumbaugh et al.

Rumbaugh et al. [Rumbaugh, et al. 91], capture the dynamic behaviour of an object-oriented system using state diagrams based on Harel's [Harel 87] state charts.

A state diagram is constructed for each object in the system and contains:

- A collection of guarded transitions corresponding to events (directed arcs in figure 7)
- A collection of states (ellipsis in figure 7).

Figure 7: State diagram for a vending machine

A transition is defined as:

`event (arguments) [guardCondition] / action`

where *event* is the name of an event. Events represent interactions between objects. (*arguments*) is a collection of arguments carried with the event. *guardCondition* is a predicate expression which must evaluate to true in order for the transition to be taken. Transitions with guards must have the appropriate event arrive and satisfy their guard condition. *action* specifies action code to be performed prior to arriving in the next state. All transitions leaving a state must correspond to different events. In figure 7 the transition *cancel / refund coins* indicates that if the event *cancel* is received while in state *Collecting Money*, the vending machine generates *refunds coins* and moves into the *Idle* state. This transition does not contain any guard and the event does not carry any arguments. Transitions are not obligated to specify an event. For example, the transition *[change = 0]* indicates that if the guard condition *change = 0* becomes true while in the middle state of figure 7, the, the object will enter the *Collecting Money* state.

States contain actions performed on entry to the state. For example, on entry to the middle state, the action,

`do: test item and compute change`

is performed. Actions are prefixed by the keyword *do:*.

States can be arranged into a generalization hierarchy represented using contours. Substates inherit the transitions of their superstate. Figure 8 shows an example of a superstate called *Forward*. *First*, *Second* and *Third* are types of *Forward* states. If the transmission object is in the first state it implies it is in a type of *Forward* state and hence the event *push N* will cause a transition to the *Neutral* state.

Actions in states and on transitions can generate events to other objects using a *send event* operator. This is similar to Schlaer and Me'lor's *generate:* construct.

Figure 8: State diagram with contours

2.2.1 Discussion

Rumbaugh et al. state notation has several improvements over Schlaer and Mellor. First, it adds the ability to associate guards with transitions, and to decompose transitions and actions into substate diagrams. Second, states can be organized into abstraction hierarchies. These additions assist in simplifying the complexity of a diagram associated with a given object thus making it easier to comprehend.

Rumbaugh et al., like Schlaer and Mellor, describe a particular execution of the system in terms of the exhibited events. Events occurring during the execution are captured in an event trace (see figure 10) similar to Schlaer and Mellor's threads. Event traces are expressed by Rumbaugh et al. using an event trace diagram (see figure 9). These diagrams are more formal than the thread diagrams of Schlaer and Mellor.

Rumbaugh et al. extend the state diagram notation to explicitly identify events sent to other objects by transitions (see figure 11). Unfortunately, a similar diagramming mechanism is not used to identify events generated by actions. Consequently, interactions can become obscured by the actions.

Rumbaugh et al. lack a big picture describing the overall system behaviour. This could partially be alleviated by using Schlaer and Mellor's OCM diagrams.

Figure 9: Event trace diagram

caller lifts receiver
 dial tone begins
 caller dials digit(5)
 dial tone ends
 caller dials digit(5)
 caller dials digit(5)
 caller dials digit(1)
 caller dials digit(2)
 caller dials digit(3)
 caller dials digit(4)
 called phone begins ringing
 ringing tone appears in calling phone
 .
 .

Figure 10: Event trace.

Figure 11: Extended state diagram notation

2.3 Bear et al.

Bear et al. [Bear, et al. 90] describe an object-oriented system using a hybrid notation combining the state charts of Rumbaugh et al. and the object communication model diagrams of Schlaer and Mellor. The behaviour of a system of objects is described using a configuration diagram and a collection of state charts, one for each object.

2.3.1 Configuration Diagrams

A configuration diagram (see figure 12) identifies the objects (boxes) participating in the system, the services they provide (solid lines) and the services they require (dashed lines). Object interactions are denoted by lines connecting required services with provided services. Figure 12 illustrates an alarm clock system consisting of three objects, an alarm clock, a notification window and a system clock. The alarm clock requires the service *open window* as depicted by the dashed line leaving it. The dash line terminates as a solid line at the window object. Hence, the window object provides the *open window* service. The alarm clock object defines several services not used by any other object in the system. These services are: *set*, *cancel*, *stop*, *time of day* and *alarm time*.

2.3.2 State Diagrams

State diagrams (see figure 13) use Harel's state chart notation. A state diagram consists of a collection of states and the transitions between them. Figure 13 shows the state diagram for the alarm clock of figure 12. This state diagram utilizes Harel's "AND composition" and contours. The "AND composition" signifies the left and right portions of the diagram are executed independently of each other. In the case of the alarm clock, the time is updated in the left portion of the diagram independent of whether the alarm is on or off. Contours denote hierarchies of states

similar to those described by Rumbaugh et al.

Figure 12: Configuration diagram for an alarm clock system

Transitions between states have the form:

```

initialState -> finalState
{guardCondition}
/requiredService(inputParameters | outputParameters)
{postCondition}
  
```

where *guardCondition* is a predicate which must hold in order for the transition to be taken. Transitions describe the conditions under which the services of one object are required. The transitions for the alarm clock are shown in figure 14.

Figure 13: State diagram for an alarm clock object

Examining figure 13 and figure 14, there are four transitions specifying services required by the alarm clock. The required services are *openWindow* and *closeWindow* which must be provided by object *W*, and the operation *time* which object *C* must provide. The configuration diagram in figure 12 specifies the objects corresponding to *W* and *C* are Window and Clock respectively.

- $\text{timeupdate} \rightarrow \text{timeupdate: } \{\text{true}\} / \text{C.time?}(|t)\{\text{timeofday} = t\}$
- $\rightarrow \text{timeupdate: } \{\text{timeofday} = 0\}$
- $\text{alarm off} \rightarrow \text{alarm on: } \{\text{true}\}\text{set}(t)\{\text{alarmtime} = t\}$
- $\text{alarm on} \rightarrow \text{alarm off: } \{\text{true}\} \text{cancel } \{\}$
- $\text{quiet} \rightarrow \text{ringing:}$
 $\{\text{alarmtime} \leq \text{timeofday} < \text{alarmtime} + 1.5\}$
 $/\text{W.openwindow } \{\text{finish} = \overline{\text{timeofday}} + 60\}$
- $\text{ringing} \rightarrow \text{quiet: } \{\text{timeofday} \geq \text{finish}\}$
 $/\text{W.closewindow}\{\}$
- $\text{ringing} \rightarrow \text{quiet: } \{\text{true}\} \text{stop } / \text{W.closewindow } \{\}$

Figure 14: Some transitions for an alarm clock object.

The first transition of figure 14 illustrates the mechanism for passing parameters to a requested service. Input and output parameters sent with the service request specify the servicing object should either use or provide the variables in response to the request. For example, $/\text{C.time?}(|t)$ specifies there are no input parameters passed to the *time?* operation but, object C must place a value into the variable *t*. In other words, C must output *t* in response to receiving a request for the *time?* service.

Invariants are captured by specifying a state and the predicate which must hold while in the state. For example,

- $\text{quiet: } \text{timetoalarm} = \text{alarmtime} - \text{timeofday}$

describes an invariant specifying when the alarm will trigger.

The state diagrams of both Rumbaugh et al. and Bear et al. support the identification of what Bear et al. term “observer” services, that is, events (service requests) which do not cause a change of state. In both approaches the observer services are written inside the state. For example, the event *timeofday* in state

timeupdate of figure 13 does not cause a transition. The service is provided to clients but the alarm clock remains in the *timeupdate* state.

2.3.3 Discussion

Configuration diagrams are similar to Schlaer and Mellor's object communication model diagrams. The principle difference is in the intent of their respective authors. OCM's are intended for use during the analysis phase and are viewed by Schlaer and Mellor as a mechanism for capturing real world events. Configuration diagrams capture the exchange of actual service requests and as such specify all services provided by objects in the system.

There are two important aspects of the state diagrams when compared to Rumbaugh et al. First, the states of Bear et al. do not contain actions. Actions are part of the services and elicited when interactions (service requests) are brought about through the traversal of transitions. This approach results in a simpler diagram that focuses on the high level behaviour of the object. The details of how an object responds to an event or the fine grain details of the behaviour within each state are left to the implementation of the operations. The object's behaviour can therefore be captured without completely abandoning the use of classes to define individual operations. In other words, existing class and operation descriptions can be used while augmenting them with a graphical view. This is beneficial because it builds on the user's existing notation and knowledge.

Second, service requests of Bear et al. specify the object providing the service; Rumbaugh et al. employ a broadcast style approach where an event is generated and those objects whose state diagrams can accept the event do so. Communication in [Bear, et al. 90] is point to point. A potential problem with this approach is that an object's state diagram can become forever coupled to the objects it interacts with. This means the object cannot be reused since it requires the objects it explicitly identifies in its service requests. To overcome this problem, Bear et al. use aliases

within state diagrams. For example, the state diagram of the alarm clock uses **C** in place of **Clock**. It is the responsibility of the configuration diagram to identify the objects the aliases are eventually bound to. In the case of the alarm clock, a second configuration diagram could potentially replace the **SystemClock** with a **DigitalClock**, **C** then becomes bound to the **DigitalClock**. The drawback to aliases is they can potentially reduce the readability of the diagram.

Bear et al. highlight object interactions while providing the flexibility to form alternative system compositions. However, they stop short of making configurations be first class objects. They do not support the inheritance or composition of configuration diagrams. Finally, Bear et al., like several other notations, use event traces to describe a particular execution of the system.

2.4 Wirfs-Brock et al.

Wirfs-Brock et al. [Wirfs-Brock, et al. 90b] describe an object-oriented system by identifying *collaborations* (interactions) between objects and groups of objects. Responsibilities are defined for each object in the system. A *responsibility* is the obligation of one object to provide another object with a given service. Responsibilities are grouped into disjoint sets called *contracts*. An object is thus described in terms of the contracts it supports. Subclasses support the contracts defined by their superclass.

2.4.1 Collaboration Graph

A *collaboration* exists between two objects when, one object requests a service and the other provides it. Collaboration graphs (see figure 15) capture the collaborations between objects (rectangles) in the system. Collaborations are specified using directed arcs. In figure 15 the ATM object uses a service provided by the transaction object. A semi circle (with a number) at the boundary of an object identifies the specific contract supplied by the particular type of object. Thus, ATM specifies

it requires the services supplied in contract number 8 from the transaction object.

Class inheritance is modelled using contours. In figure 15, the transaction class has four subclasses, *BalanceInquiry*, *DepositTransaction*, *WithdrawalTransaction*, and *FundsTransfer*. Subclasses inherit the contracts of their superclass. Hence, the four subclasses of transaction support contract 8 defined by transaction. Further examination of figure 15 shows the class *UserInteraction*, does not support any contracts however, its three subclasses, *Form*, *Menu* and *UserMessage*, supply contracts, 4, 6 and 9 respectively. Arcs directed towards the *UserInteraction* class are therefore terminated at the individual subclasses. Finally, classes can provide private responsibilities which cannot be requested by other objects.

2.4.2 Subsystem Diagram

An object-oriented system is broken into a set of subsystems. A *subsystem* is a group of classes and other subsystems that cooperate to support a set of contracts. If a subsystem supports a given contract, it does **not** imply all of its members support the contract. Instead, it signifies the contract is serviced by at least one member of the subsystem.

Subsystems are described using a subsystem diagram (see figure 16). A subsystem diagram represents the subsystem in terms of nested contours with each contour defines a class or subsystem and the contracts it supports. Subsystem diagrams also specify collaborations among the subsystem's participants. Semi circles at the boundary of a subsystem denote contracts supported by the subsystem. A directed line is drawn from the semi circle to the class or subsystem supporting the contract. For example, in figure 16 the financial subsystem supports contract number 8. Contract number 8 is supplied by the transaction class and its subclasses as illustrated by the line connecting the subsystem to its transaction member.

Figure 15: Collaboration graph for an ATM system

2.4.3 Discussion

Wirfs-Brock et al. provide a high level view of the system free of control information exhibited by the previous three notations. The approach is more interaction based than either Rumbaugh et al. or Schlaer and Mellor, and focuses on objects and groups of objects cooperating to accomplish tasks. Unfortunately, the notation of Wirfs-Brock et al. captures a static snapshot depicting the system if all interactions were to be triggered at once. It does not show overall flow of the system, and there is no high level view of the patterns of communications that arise.

A form of information hiding is accomplished by restricting objects outside a subsystem to use only those contracts specified at the subsystem boundary. This effectively makes the interactions within the subsystem be private. This is similar to interactions within an object being private through the use of private responsibilities. Wirfs-Brock et al. support composition hierarchies by allowing subsystems to contain other subsystems. This addresses a problem identified with the configuration diagrams of Bear et al. However, subsystems cannot be arranged into an inheritance hierarchy. Thus, subsystems are essentially hierarchical collaboration graphs. Rumbaugh et al. [Rumbaugh, et al. 91, pp. 199-200] identify the need to break a system into subsystems in a manner similar to Wirfs-Brock et al. However, the notation of Rumbaugh et al. does not support subsystems.

A problem noted by Wirfs-Brock et al. is the possibility a collaboration graph ends up looking like "spaghetti" [Wirfs-Brock, et al. 90b, pp. 143] with lines going in all directions. Diagrams of this type are difficult for human readers to comprehend. Wirfs-Brock et al. suggest that such diagrams are the result of a poorly structured system and not a problem with the notation.

Figure 16: Subsystem diagram for a portion of the ATM system

2.5 Pernici

Pernici [Pernici 89] describes an object in terms of the *roles* it can play. A class is defined by the collection of role types it supports. A role consists of:

- The role's type name.
- Attributes of the object's state which pertain to the role.
- A list of messages the object can receive while in the role.
These messages are prefixed by $\leftarrow$.
- A list of messages the object sends while in the role.
These messages are prefixed by $\rightarrow$.
- Transition rules specifying how the behaviour of the object changes while in the role.

An instance of a given role type is called a *role*. An instance of a class may contain many roles but it does not necessarily contain an instance of each role type. Roles are created and destroyed as needed by the object. An object can simultaneously be in multiple roles, and a role type can be instantiated many times. A special role type called the *base role type* describes behaviour common to all roles and defines rules to coordinate the roles. A role can be temporarily suspended. A suspended role does not receive or send any messages until a *resume* message arrives. On receipt of the resume message the role resumes normal processing.

Transition rules defined by a role have the form:

```
rule : predicate [(,predicate)*]
      => predicate [(,predicate)*]
```

where

- [] denotes optional
- ()* means the term in parenthesis is repeat zero or more times
- , denotes conjunction (AND)

Pernici supports two main predicates:

- To test if *object* is in the in role *roleName* and in state *roleState* of that role:

```
state ([object,] [roleName,] roleState)
```

- To test if the service has been requested by *object* who is in the role *roleName*.
The message can optionally specify arguments allowing them to restrict the acceptance of a message to those situations in which a given data value is passed:

```
msg ([object,] [roleName,] message)
```

The predicate *state()* is special and is true no matter what state the role is in. Consider the following transition rule:

```
state (running) , msg (halt)
=> state (stopped), msg(object1 outOfService)
```

This rule specifies, that if the role is in its *running* state and a *halt* message is received, then the role enters the *stopped* state and sends the message *outOfService* to *object1*.

Pernici also defines operations for:

- creating and removing roles
- suspending and resuming roles
- testing whether an object is in a role, or not
- testing if a role is of a specific role type

2.5.1 Diagrams

The relationship between roles is depicted using a role transition diagram. In figure 17 the *accident* role can be instantiated at any time using the add-role message. Once in the accident role, messages are sent to either suspend or resume the transportation role. The destruction role can be instantiated at any time and sends messages to the other roles to terminate them.

Figure 17: Role transitions of a car

The behaviour within a role is represented using a state transition diagram called a role state diagram (see figure 18). Messages sent and received while transitioning between states are denoted by $\rightarrow message$ and $\leftarrow message$ respectively. In figure 18 the object sends the message $\rightarrow expired$ to another object while traversing between the two states.

Figure 18: State diagram of a role

2.5.2 Special Rules

Invariants maintained by the object are expressed using a special rule called a *constraint*. Constraint rules specify the condition and the constraint which must hold if the condition evaluates to true. For example,

```
constraint ( (fuel = 0) => not state(running))
```

specifies, that if a vehicle has no fuel it cannot be running.

An *equivalence* operator can be used to declare that two messages are synonyms for each other when in a given role. For example, the accident role of figure 18 can be terminated by the receipt of either a *ready* message or a *terminate-role* message. This is expressed with the following equivalence rule:

```
equivalent (ready terminate-role)
```

Pernici also provides operations to ignore inherited rules, and block out messages. For example,

```
constraint ((fuel = 0)
           => forbidden-op (transportation, switch-on))
```

specifies that if the vehicle is out of fuel, the *switch-on* operation of the role *transportation* is forbidden.

2.5.3 Discussion

Pernici's role types can be instantiated multiple times, however there is no support for coordinating multiple instances of the same role type. Pernici illustrates multiple instances using an insurance example. In the example the insurance role is instantiated twice, once to describe the current insurance and a second time to describe future insurance that is being negotiated. Unfortunately, operations in Pernici's notation do not support the identification of the individual instances and do not provide for the coordination of the instances. Furthermore, the motivating example for multiple role instance is itself dubious in that it would be more appropriate to model the situation using multiple relationships between the car and insurer objects.

Pernici's approach illustrates a problem with state transition diagrams. STD's tend not to convey a feeling for the purpose of an object unlike Pernici's roles which help to define an object's purpose in life. STD's could lessen the problem by using contours to denote roles, however, this would potentially become confusing and visually messy, especially if the diagram also supported state inheritance or Harel style contours. Furthermore, such a partitioning would not capture the first class nature of roles³.

Pernici describes an object as being in a role. A more appropriate description is: the object is exhibiting the behaviour associated with a role. This distinction reduces the impression a role is merely a state or associated with one of the object's attributes being in a specific state.

Separating constraints from other transition rules is different than other approaches which use guarded transitions. A benefit of this approach is that the constraints are made explicit and can be refined, added or ignored by subclasses. This allows one to determine if a subclass is only altering the set of constraints or if the states are being changed. Thus, Pernici supports a finer grain of inheritance

³First class implies the roles are objects too.

then provided by the other notations.

Determining the changes introduced by a subclass can become complex when transition diagrams are used. Pernici's notation does not suffer from this problem. However, the mechanism for replacing a transition rule defined in a superclass can itself result in confusion. Replacing a rule requires deletion of the parent rule and then the creation of a new rule. The inability to designate a rule as a refinement of an inherited rule can make an inheritance hierarchy difficult to understand.

Pernici does not support compositions of objects and the maintenance of invariants between objects. In addition, the notation makes it possible for one object to specify a transition on another object. This capability is undesirable since it violates encapsulation and only serves to scatter an object's behaviour throughout the system. It is likely this is an oversight in the notation.

Pernici's use of the notation illustrates a potential problem. It reveals that roles which are intended as first class, are not themselves encapsulated. Consequently, constraints can be specified on one role by another role and, constraints that should be local to a role can be incorrectly defined as global rules. An object's constraints can potentially become difficult to grasp.

Pernici focuses only on those messages which cause a state change. This includes messages sent to other objects. Clearly, all messages sent externally should be included since it is indeterminable from one object's perspective whether the message will cause a change of state in one or more roles of the object receiving the message. Another drawback is the use of role constructor and destructor messages. These special messages should be reserved for use within an object when referencing itself. Other objects should be forced to use normal messages, otherwise, one object knows too much: an object's role structure is exposed to other objects.

The role-centered view has the disadvantage that it is difficult to ascertain when a given message can be accepted. To determine this, a client must examine all of the rules referencing the message. The client must also pay particular attention

to constraints involving the message and rules specifying the removal of rules or messages. Consequently, clients need to know the internal role structure of an object in order to completely determine the applicability of its messages.

2.6 Helm et al.

Helm et al. [Helm, et al. 90] organize an object-oriented system into groups of interacting objects called *contracts*. A contract defines:

- a set of participating objects
- any invariants maintained by the contract
- a specification of how the contract is instantiated

For each of the contract's participants the following is specified:

- Operations provided by the participant and their behaviour. These are the causal obligations of the participant. Operation behaviour is specified using a formal specification.
- Variables maintained by the participant and their type.

A contract can specialize another contract by using the *refinement* statement and specifying the name of the contract to refine. Refinement allows:

- New instance variables and operations to be added to participants.
- A participant's causal (behaviour expected of a given operation) obligations to be refined.

A contract can be composed of other contracts. This is accomplished by using the *include* statement and specifying the name of the contract to include. Both the *include* and the *refines* statements allow synonyms to be specified for participants of the contract being included or refined.

The behaviour of a participant's operation is described in a formal algebraic notation supporting message passing, if-then-else and looping. An operation's specification has the form:

$$\text{operationName}(\text{arguments}) \mapsto \text{code}$$

where *code* supports the following constructs:

- *expression1 ; expression2* – specifies *expression1* is performed before *expression2*
- *anObject* $\rightarrow$ *msg* – indicates the message *msg* is sent to the *anObject*
- Δ – specifies the setting of an instance variable
- if-then-else and for loop constructs
- $\parallel$ – denotes no specific ordering
- $\langle \text{operator variable} : \text{selectionCondition} : \text{expressionToPerform} \rangle$ indicates the repetition of *expressionToPerform* on all objects satisfying *selectionCondition*. *operator* denotes the order operations are to be performed in
- $\{ \dots \}$ denotes a predicate that must be made true

Figure 19 shows the subject-view contract from [Helm, et al. 90]. This contract describes a system in which a subject object containing some data is graphically represented in a number of ways (views). The subject-view contract defines two participants, subject and view. Examining the *subject* participant we see that it supports five operations. The notify operation:

$$\text{Notify}() \mapsto \langle \parallel v : v \in \text{Views} : v \rightarrow \text{Update}() \rangle$$

specifies that on receiving the *notify* message, the subject sends the message *Update* to all of the views. The above statement also specifies (by $\parallel$) that there is no specific ordering to the way in which the views are sent the messages.

Within a contract, the term “A **reflects** B” represents an informal relationship between A and B. The informal relationship identifies, in high level terms, the existence of a relationship but does not specify the finer details about the relationship. In figure 19 the *draw* operation specifies that the *views reflect the subject value*.

Invariants are expressed as

expressionToSatisfyInvariant $\mapsto$ invariant

The subject-view invariant,

Subject.SetValue(val) $\mapsto$ $\langle \forall v : v \in \text{Views} : v \text{ **reflects** Subject.value} \rangle$

specifies that sending the message *SetValue* to the subject participant will cause the invariant condition to become true. The invariant condition requires that all views **reflect** the subject value.

Contracts also specify how the contract is to be instantiated. The subject-view instantiation clause,

$\langle \parallel v : v \in \text{Views} : \langle \text{Subject} \rightarrow \text{AttachView}(v) \parallel$
 $v \rightarrow \text{SetSubject}(\text{Subject}) \rangle \rangle$

specifies that in order to instantiate the contract; for each view the subject must be attached to the given view and vice versa. The operations *AttachView* and *SetSubject* are used to connect the objects to each other.

contract *SubjectView*

Subject **supports**[

value : *Value*

SetValue(val : Value) $\mapsto \Delta value\{value = value\}; Notify\{\}$

GetValue() : *Value* $\mapsto$ return *value*

Notify() $\mapsto \langle ||v : v \in Views : v \rightarrow Update() \rangle$

AttachView(v : View) $\mapsto \{v \in Views\}$

DetachView(v : View) $\mapsto \{v \notin Views\}$

]

Views : *Set(View)* **where each View supports**[

Update $\mapsto Draw()$

Draw $\mapsto Subject \rightarrow GetValue()\{Views \text{ reflects } Subject.Value\}$

SetSubject(s : Subject) $\mapsto \{Subject = s\}$

]

invariant

Subject.SetValue(val) $\mapsto$

$\langle \forall v : v \in Views : v \text{ reflects } Subject.value \rangle$

instantiation

$\langle ||v : v \in Views : \langle Subject \rightarrow AttachView(v)||$

$v \rightarrow SetSubject(Subject) \rangle \rangle$

endcontract

Figure 19: An example contract for a subject and views.

The contracts of Helm et al. are orthogonal to classes with the emphasis on capturing the interactions. Contracts are mapped to an implementation using conformance declarations in which a participant declares which classes implement a participant's operations. A participant's operations can potentially be spread across a hierarchy of classes.

2.6.1 Discussion

Helm et al. focus attention on groups of interacting objects. Inter-object dependencies are expressed in terms of the messages exchanged and any invariants they cooperate to maintain. Using the refinement and include operators it is possible to form abstraction hierarchies and compositions of groups of objects. Thus, unlike other approaches, the focus is moved from being primarily on the behavior of an individual object to the interactions of a group. Unfortunately, Helm et al. embed the object interactions in operation specifications. It is difficult to determine the thread through a contract especially when synonyms are introduced by the include and refines constructs. The added complexity introduced by synonyms could be partially reduced through the use tools which performed name substitutions for synonyms.

Helm et al. identify a property of interacting objects, namely, specialization of a group of interacting objects often requires the specialization of more than one of the participating objects. Contracts localize these changes into a single unit, making it easier to determine the net changes made.

Helm et al. do not permit the refinement of an operation to invoke the inherited behaviour (eg. Smalltalk's *super*). Also, their notation can be confusing at times. In particular, specifying or trying to understand a group of objects, can be difficult requiring one switch between the programming language and the notation of Helm et al. The algebraic notation may appeal to individuals with a formal background.

Holland [Holland 92] extends the notation of Helm et al. by requiring participants to explicitly declare whether an operation is visible outside a contract. This is beneficial since it allows a contract to hide operations provided by its participants.

A drawback to the approach of Helm et al. is it abstracts away the behaviour of the system and forces one to switch between classes and contracts. The problem becomes more pronounced when the mapping between classes and contract participants (defined using a conformance declaration) is not one to one. The severity

of this is hard to assess without building a medium-sized system and its associated contract(s). If the problem is too severe then the essence of the system could be lost – an undesirable situation as Harel [Harel 92] and Brooks [Brooks 87] have observed.

2.7 Summary

In this chapter we examined several notations for describing objects. For the most part these notations focus on the behaviour of an individual object. The interactions within the system are captured in a static manner depicting the system's behaviour if all events were to simultaneously occur.

Several notations describe the use of event traces. An event trace captures the sequence of events resulting from a particular execution of the system. The notations presented here have used event traces in a post-execution manner; using a trace to show how the system behaves under a specific situation or to determine whether a specific sequence could have resulted from the system.

The contract notation of Helm et al. is the only notation which focused specifically on groups of interacting objects, and permitted the inheritance and composition of groups. The subsystems of Wirfs-Brock et al. only partially address the group aspect.

The notation of Helm et al. illustrates the difficulties involved in comprehending an unfamiliar notation. With respect to the requirements identified in Chapter 1, notations such as Helm et al. risk drawing the user's attention away from comprehending the system and place it on understanding the notation. We are not advocating the abandonment of formal notations such as Helm et al. but rather in trying to understand the high level aspects of a system, the notation should be kept as simple and familiar as possible.

Figure 1 presents a summary of the notations. The table identifies those aspects of each notation that are relevant with respect to the requirements identified in Chapter 1.

In the next chapter we present the PEGs notation and some of the concurrency literature it owes its origins to.

Chapter 3: PEGs

3 The PEGs Notation

The behaviour of an object-oriented system is characterized by its objects and their interactions. Notations presented in Chapter 2 focus on the behavior within an object. These notations limit their description of the overall system behaviour to a static snapshot capturing the system at an instant in time when every interaction simultaneously occurs. Although static pictures can provide a useful view of the system, they do not fully capture a critical aspect of the system, namely, its dynamic behaviour. Unfortunately, the patterns of interactions that give a system its behaviour also increase its complexity. As Brooks [Brooks 87] and Harel [Harel 92] have noted, it is important to capture the complexity of the system that forms its essence. Hence, in an object-oriented system the patterns of interactions must be captured.

Several notations discussed in Chapter 2 use an event trace to depict the interactions of a particular execution of a system. Unfortunately, the use of event traces by these notations is limited to individual executions and does not provide a description of the interactions that can be expected of the system. In other words, they do not specify what the system will do, but rather, what it did on one occasion.

In Chapter 1 we identified the necessity of being able to form groups of interacting objects. This requirement is motivated by several factors:

- Gibbs et al. [Gibbs, et al. 90, pp. 94] and Wirfs-Brock and Johnson [Wirfs-Brock and Johnson 90a] observe that reuse typically occurs either

through the reuse of groups of objects interacting to accomplish tasks, or through the specialization of frameworks.

- Frameworks are themselves groups of classes specifically designed to be refined and used as a group [Wirfs-Brock, et al. 90b, pp. 13].
- “Classes alone do not provide enough structure for large systems” [Wirfs-Brock and Johnson 90a, pp. 122].

Several notations (see figure 1, pp. 9) in the previous chapter divide a system into groups or subsystems consisting of cooperating objects. The approach of Helm et al. is significant in that it focuses attention on groups of interacting objects and supports the inheritance and composition of those groups. In general, the other notations use groups and subsystems as a conceptual mechanism for organizing the system.

In this chapter we presents the PEGs notation. PEGs achieves the requirements set out in Chapter 1. Some of the more important requirements are:

- The partitioning of the system into groups of interacting objects.
- The treatment of groups as first class objects and the support for their inheritance and composition.
- The description of the **typical** patterns of interaction that form the essence of the system.
- The ability to identify objects that provide abstract behaviour.

PEGs employs path expressions [Campbell and Habermann 73] to describe the patterns of interaction exhibited by the system. The notation supports the inheritance and composition of paths. Interacting objects are formed into groups which can be arranged into inheritance hierarchies or combined to form more complex

groups. Groups, their participating objects and the operations supported by those objects can either be abstract or concrete.

PEGs captures the essential⁴ behaviour of a system; it is not intended as a comprehensive or formal method notation. As [Schlaer and Mellor 92], [Lalonde, et al. 91] and [Lalonde 92] have noted; although an object can communicate with any other object, typically only a few patterns of communication are observed. PEGs uses path expression to capture these important patterns of interaction that form the essence of the dynamic behaviour of the system.

PEGs owes its origins to the path expression work of Campbell and Habermann [Campbell and Habermann 73] and to the ideas of Helm et al. [Helm, et al. 90].

The PEGs notation presented in this thesis is not intended to be a design methodology nor a formal specification, instead it aims to achieve a common ground between the two. In our investigation we focus on traditional sequential object-oriented systems, real-time and concurrent systems are beyond the scope of this thesis. Also, it is assumed the objects to describe already exist.

In the remainder of this chapter, we first present background material with respect to path expressions, and we motivate the use of path expressions to describe object interactions. Subsequent to the background material, Smalltalk/V's pane/dispatcher/model framework is reviewed. This is followed by a description of the PEGs notation augmented with examples based on the pane/dispatcher/model framework. Lastly, the PEGs syntax is formally defined using a Backus Naur Formalism (BNF).

3.1 Path Expressions

Path expressions (paths) were first introduced by Campbell and Habermann [Campbell and Habermann 73] and subsequently generalized by Lauer and Campbell [Lauer and Campbell 75]. Campbell and Habermann use path expres-

⁴The important aspects of the behaviour.

Operation	Meaning
A,B	Either A or B is performed.
A;B	A is performed and then B.
{A}	Any number of processes ⁵ can perform A.
()	Parenthesis enclose parts of an expression that are treated as a single item.

Figure 20: Path expression operators and their meaning.

sions to synchronize concurrent processes. A path defines a set of permissible sequences of operations on a known set of objects. In other words, a path expression defines a set of possible traces. More formally, a path expression can be viewed as a grammatical object generating a trace language [Shields 79]. To paraphrase Campbell and Habermann [Campbell and Habermann 73, pp. 91]; The notation is based on regular expressions, which provides a familiar framework and allows path expressions to be represented as finite state machines.

Campbell and Habermann's path expression notation consists of the operations shown in figure 20.

The path `PATH P1,(P2;P3) END` specifies either P1 is performed by at most one process or P2 and then P3 are performed by at most one process. The expression between `PATH` and `END` is repeatedly executed. The path `PATH P1,{P2;P3};P4 END` specifies P1 is performed by one process and then the sequence of performing P2 and then P3 is performed by any number of processes. When no more processes are performing the sequence P2;P3 then P4 is performed by at most one process.

Bruegge and Hibbard applied path expressions as a high level debugging mechanism. For example, the path expression `PATH P1;P2 END` can be used by a debugging facility to specify the system should halt once P1 and P2 have occurred in that order. Bruegge and Hibbard's perspective on path expressions is of an operational nature whereas Campbell and Habermann have a synchronization perspective. In

⁵The term "process" is used in the manner described in [Campbell and Habermann 73].

Operation	Meaning
A ; B	A is followed by B.
A B	Either A or B is performed.
A*	A is repeated zero or more times.
()	Parenthesis enclose parts of an expression that are treated as a single item.

Figure 21: Bruegge and Hibbard's path expression operators and their meaning.

other words, Campbell and Habermann are concerned with synchronizing processes and preventing invalid sequences of operations. Bruegge and Hibbard are primarily concerned with whether or not an operation has been invoked.

Bruegge and Hibbard's [Bruegge and Hibbard 83] path notation is more characteristic of the traditional style of regular expression notations than Campbell and Habermann's notation. Their notation is shown in figure 21.

The following Bruegge and Hibbard path expression defines the order in which operations are performed on a file:

```
PATH open ; (read | write)*; close END
```

The above path expression specifies the open operation occurs first, followed by any number of read or write operations after which a close operation ultimately occurs. The expression within PATH and END is repeatedly executed.

3.2 Predicate Path Expressions

Predicate path expressions [Andler 79] define a predicate condition which must hold in order for the path to apply. To illustrate, consider the following path (expressed using Bruegge and Hibbard's notation):

```
initializeStack;  
  ( (stackSize < MAX): push  
    | (stackSize > 0): (pop | top)  
  )*
```

This path expression describes the ordering of operations on a stack. The path specifies *initializeStack* is performed first, followed by any number of *push* operations provided the condition *stackSize < MAX* is true or; *pop* or *top* operations if *stackSize > 0*. In other words, the path expression (pop | top) is only permitted if *stackSize > 0*.

Path expressions belong to the set of languages known as regular and as such are recognizable with a finite state machine. Predicate path expressions reside in the domain of context-free languages. Context free languages are more expressive than regular languages, but are also more complex requiring a finite state machine controlling a push down stack and associated rules for their recognition. Barret et al. [Barret, et al. 79] provide a detailed discussion of the various classes of languages and the automata for recognizing them.

3.3 Procol and Ada Packages

Path expressions have been applied to object-oriented systems as a mechanism for describing the ordering of operations within an individual object.

Goldsack and Moreton [Goldsack and Moreton 82] use path expressions in a manner comparable to Campbell and Habermann. A path expression is associated with each Ada package. The path expression specifies the synchronization of the

operations described in the package. Path expressions in [Goldsack and Moreton 82] allow one to specify the number of processes which may simultaneously perform a given path expression. Although the notation is different than Campbell and Habermann's, the intent is fundamentally the same.

Procol [Van Den Bos and Laffra 89] uses path expressions to define a protocol for each object. For example, the Procol protocol for a bank account object is:

```
Creator(amount) -> Open;
( Creator() -> Inquire
  | Creator(amount) -> deposit
  | (balance>0) : creator(amount) -> withdraw
)*;
Creator() -> close
```

The above path expression specifies the creator object first sends the *open* message with the amount as an argument. The creator subsequently sends *inquire*, *deposit* or *withdraw* messages any number of times ultimately followed by the *close* message. Procol supports a message-sending variation on the predicate path expression theme: Procol guard conditions apply to a single operation. Consequently, Procol uses predicates as operation preconditions rather than as preconditions for an arbitrary path. In the above path expression, $(\text{balance} > 0)$ is the guard for sending the withdraw message. Procol describes inter-object communication only to the extent of specifying which object can perform an operation. In the above example, the creator object performs all of the operations.

The previous sections provided a summary of path expressions and their origins. We now motivate the use of path expressions to capture object interactions.

3.4 Why Path Expressions?

Bruegge and Hibbard observed path expressions, as described by Campbell and Habermann, are valuable for two reasons:

1. They provide a convenient method of separating the behaviour of individual operations from the behaviour of the collection of operations when they cooperate.
2. They provide a terse form of expressing the interaction between operations.

These benefits can be realized at a higher level than originally intended. Path expressions can be used to separate and capture the interactive behaviour at the level of objects and groups of objects, instead of at the lower level of operations.

PEGs uses path expressions to describe object interactions. This approach is unlike the notations of Chapter 2 and the Procol notation described above which focused primarily on individual objects and their own dynamic behaviour. Within PEGs, path expressions model "essential interactions". Traditionally, path expressions have been used to identify the sets of valid sequences. PEGs uses path expressions to identify the typical sets of interactions. Hence, it captures the essence of the system rather than providing a complete formal description.

The use of path expressions has several inherent benefits:

- They are simple, easy to understand and the rules for their construction are well known. This results from their being based on regular expressions, a familiar framework [Campbell and Habermann 73].
- The considerable amount of research into path expressions can be reused.
- The similarity of paths to the traces they generate also holds promise in the area of reverse engineering. Reverse engineering begins with the implementation and derives, possibly with manual assistance, a specification of the

system. In the presence of rapid prototyping, and with the growing number of design notations, it is becoming important to be able to regenerate specifications [Lalonde 92]. It is feasible that tools such as those under investigation by Lalonde could generate path expressions as part of the system specification or as an intermediate step to assist users in arriving at a comprehensive specification.

- Finally, path expressions force the system to be viewed in terms of its overall behaviour instead of focusing on the behaviour of isolated objects.

Before presenting the details of the PEGs notation we first review the framework that will form the basis for examples presented in the remainder of this chapter. This framework, the pane/dispatcher/model framework is a collection of classes that are intended to be refined or reused to build windows.

3.5 Pane/Dispatcher/Model Framework

In the following sections we present the PEGs notation using examples based on Smalltalk/V's pane/dispatcher/model framework. We now provide a brief review of the framework. Refer also to [Digitalk 88].

Windows in Smalltalk/V are divided into regions called *Panes*⁶. The processing of input within a region is controlled by a *Dispatcher* associated with the pane. A special pane called the *TopPane* is responsible for coordinating all of the panes within a window. The TopPane is also responsible for the label bar at the top of the window, and for ensuring that all of the window's panes occupy their assigned regions of the window. The TopPane-Pane relationship is analogous to an egg carton with the TopPane providing a labelled container (eg. grade A eggs) and the panes (eggs) being placed into the container as needed.

Formally, a window consists of:

⁶The names of the classes are used to clarify the particular classes involved.

- A TopPane and its associated TopDispatcher. The TopPane is responsible for displaying the window's label and any icons on the left and right sides of the label.
- A collection of Panes and their associated Dispatchers. These panes are often referred to as subpanes.

Messages relating to opening, closing, moving, resizing, activating, and deactivating the window are sent to the TopPane/TopDispatcher. If appropriate, the TopPane forwards the messages to the subpanes.

When the user selects a window and moves the cursor into the window, the TopPane/TopDispatcher determines which area the cursor is in. The Pane/Dispatcher occupying the particular region is instructed to begin processing input (by sending the Dispatcher the message *processInput*). When the cursor leaves the area, the Pane/Dispatcher returns control to the TopPane/TopDispatcher which repeats the search process. The TopPane/TopDispatcher only processes input if none of the subpanes contain the cursor. Dispatchers use a polling strategy to process input.

Panes are controlled by a *model* object. The model is responsible for ensuring the Pane displays the correct (ie. current) information. The model also provides the information to be displayed. Panes are responsible for informing the model when the user performs an important action.

Example: If the model wants the Pane to refresh the displayed information, the model sends the message *update* to the Pane. On receipt of this message the Pane asks the model for the information to be displayed and then refreshes the affected area of the window. If the user subsequently performs an important action (eg. modifying and saving the displayed information) the Pane sends an appropriate message to the model.

Windows in Smalltalk/V are managed by a scheduler (an instance of Dispatcher-Manager). To create a window, the TopPane is created first. Next, the subpanes are

created and added to the TopPane using the *addSubpane:* message. The window's label can be configured by sending the *label:* message to the TopPane. Similarly, the messages *leftIcons:* and *rightIcons:* are used to specify the icons to display in the left and right corners of the label bar respectively. Once the window has been created it is registered with the scheduler (ie. scheduled).

The Smalltalk/V pane/dispatcher/model framework comes with a collection of panes and dispatchers. Following is a subset of the pane/dispatcher class inheritance hierarchy. Classes (abstract and concrete) not explicitly used by the examples in this thesis have been omitted:

```
Pane
  SubPane
    TextPane
      TopPane
        Dispatcher
          TextEditor
            TopDispatcher
```

3.6 PEGs

PEGs describes an object-oriented system using two primary mechanisms:

1. path expressions and,
2. groups of cooperating objects

Interacting objects are formed into groups which can be arranged into inheritance hierarchies in a manner analogous to the subclass-superclass relationship. Interactions within a group are represented by path expressions. A group inherits its path expressions from its supergroup⁷. A group's path expressions can reference inherited paths in a manner comparable to the super notation provided for

⁷The terms subgroup and supergroup are used to convey the subclass-superclass nature of the relationship.

Operation	Meaning
A ; B	A is followed by B.
A B	Either A or B is performed.
A*	A is repeated zero or more times.
A+	A is repeated one or more times.
[A]	A occurs zero or once.
A & B	Either A is performed and then B or vice versa.
()	Parenthesis enclose parts of an expression that are treated as a single item.

Figure 22: PEGs' path expression operators and their meaning.

Smalltalk methods. Paths can be composed of other paths. Groups can replace (by redefining the path) or add to an inherited path.

In addition to containing singleton objects, a group can also be composed of other groups. The process of using other groups is essentially a part-of relationship, with the complex group using other groups as parts. The use of one group within another group implies the use of all of its constituent elements. In other words, groups are treated as a single unit when used by another group. A group can use another group multiple times.

We will now present the features of PEGs and illustrate them using examples based on Smalltalk/V's pane/dispatcher/model framework.

3.6.1 Path Expressions

PEGs' path expression operators are shown in figure 22. The notation is similar to Bruegge and Hibbard's with the addition of several new operations. These added operators are provided as convenience mechanisms and can be expressed in terms of the basic path operators (see section 3.6.6).

In traditional path terminology **A** and **B** of figure 22 refer to procedures. In PEGs, **A** and **B** specify a target object and the operation performed by it. For example, a partial PEGs path describing the behaviour of a window viewing a file

is:

```
( (aFile read; aWindow updateText)
  | (aWindow saveText ; aFile write)
)*
```

This path expression specifies either the file is read and the window's text updated or the contents of the window are saved and written to the file. The entire path can be repeated any number of times (as indicated by *).

Unlike Procol [Van Den Bos and Laffra 89], path expressions in PEGs do not specify the arguments passed to operations. This stems from the difference in what each notation is trying to capture. Procol focuses on the protocol of a given object, the ordering of invocations of its methods and the types⁸ and restrictions on its arguments. PEGs' path expressions focus on the system at a higher level. But, if required, the types and restrictions of message arguments can be obtained from PEGs specifications associated with the individual methods (described later in this chapter).

3.6.2 Groups

A group defines a collection of objects and groups that interact to accomplish a task. Groups are first class objects and have classes associated with them. A formal definition of the syntax for groups is shown in section 3.7.1. A group's declaration consists of four sections. The four sections and their associated fields are:

1. General information and declaration of the supergroup:

name

The name of the group. This corresponds to the name of the class associated with the group.

⁸The term *type* is not intended to correspond to its use in type theory, but rather to refer to the typicality of objects.

descriptiveName

A more expressive name for the group. The descriptive name is not restricted to conform to a valid class name.

description

A textual description of the group.

abstract

Identifies whether the group is an abstract group or not.

inheritsFrom

The supergroup the group is inheriting from⁹. Inheritance is examined in section 3.6.3.

2. Participating objects or other groups used by the group:

defines participants

A collection of objects whose definitions are specified by the group.

uses

Groups can be composed of other groups. The *uses* field contains a collection of other groups used by the group. Composition is examined in section 3.6.4.

secondary participants

A collection of objects used by the group but not strongly coupled to other members of the group. The kind of object typically providing the services and the services needed are specified.

synonyms

Alternative names for objects or groups defined by or used by the group.

3. Invariants maintained by the group:

⁹PEGs uses single inheritance.

invariants

Specifies, in a high level form, the invariants maintained by the group and the path expressions causing their satisfaction.

4. Path expressions describing the behaviour of the group:**behaviour**

A path expression describing the high level behaviour of the group. This path expression typically references other path expressions.

paths

A collection of path expressions providing a more detailed view of specific interactions. A name is associated with each path.

exception paths

A collection of path expressions describing the interactions associated with exception cases. A name is associated with each path.

We will now describe each of the above sections and their fields by way of the Pane/Dispatcher example shown in figures 23-27. For a formal definition of the PEGs syntax see section 3.7.1.

The first section of a group's declaration (see figure 23) specifies the name of the group and a description for the group. Two separate name fields, *name* and *descriptiveName*, are provided in order that program tools such as compilers can work with a formal syntax while still allowing the flexibility to provide a verbose descriptive name. The *abstract* field indicates whether or not the group is an abstract group. Abstract groups are intended to be specialized much in the same manner as abstract classes are intended to be specialized. Abstraction and inheritance are essential elements in promoting reuse of groups. The Pane/Dispatcher group

```
name:
    PaneDispatcher

descriptiveName:
    Pane/Dispatcher Framework

description:
    A Pane and Dispatcher cooperate to control a particular
    region of a window. Dispatchers are responsible
    for processing I/O. Panes are responsible for presenting
    the data to the user. Panes communicate with a controlling
    model from which their data is obtained.

abstract:
    yes

inheritsFrom:
    none

.
.
.
```

Figure 23: Section 1 of the Pane/Dispatcher group.

shown in figure 23 is an abstract group. Section 3.6.3 examines a subgroup of the Pane/Dispatcher group. The *inheritsFrom* field specifies the group's supergroup. The Pane/Dispatcher group is the root of an inheritance hierarchy and as indicated by the keyword *none*, does not have a super group.

The second section of a group's declaration (see figure 24) specifies the members of the group. Group members are one of the following:

- singleton objects declared by the group (*participants*)
- objects providing services to the group but not explicitly part of the group (*secondaryParticipants*)
- other groups

In addition, a collection of synonyms is specified.

Participants interact closely to define the behaviour of the group while secondary participants are loosely coupled and provide services to the group. For this reason, the declaration of a secondary participant only identifies the kind of object that is typically used by the group and known to provide the services required by the group. The form for declaring a secondary participant is:

```
participantName IS LIKE [a|an] Class
```

Examples:

```
entry IS LIKE a House  
item IS LIKE an Integer
```

Secondary participant declarations can optionally provide (using the *SUPPORT* clause) a list of messages the object is expected to supply. For example,

```
scheduler IS LIKE a Dispatcher SUPPORTING  
    scheduleWindow;  
    resume.
```

specifies that the secondary participant known as scheduler is "typically" a Dispatcher. Furthermore, only operations *scheduleWindow* and *resume* of scheduler are used.

Secondary participant declarations serve to provide clients with a starting point when considering alternatives to replace the objects typically used by the group. Without this mechanism, clients of the group would have difficulty determining the alternative kinds of objects that could be used in cooperation with the group.

If more than one kind of object typically provides the required behaviour, the secondary specification can be written as:

```
A (IS LIKE a B | IS LIKE a C)
or
A (IS LIKE a B | IS LIKE a C) SUPPORTING
  message1;
  message2;
  etc.
```

The above expression specifies A is typically either a kind of B or a kind of C.

The Pane/Dispatcher group in figure 24 defines two participants, *dispatcher* and *pane* which are instances of the classes Dispatcher and Pane respectively. The Pane/Dispatcher group does not use any other groups, but does specify four secondary participants, scheduler, topPane, topDispatcher and model. Comments are contained within double quotes ("")

Synonyms provide a mechanism for attaching alternative names to objects. Synonyms are helpful when an object is being considered from different perspectives. In the Pane/Dispatcher group, the synonym definition,

```
pane = dispatcher pane.
```

specifies that *pane* and *dispatcher pane* represent the same object. Synonyms help clarify alternative ways of describing an object and allow convenient short forms.

```
.  
.  
defines participants:  
    dispatcher IS a Dispatcher.  
    pane IS a Pane.  
  
uses:  
    none  
  
secondary participants:  
    scheduler IS LIKE a DispatcherManager.  
    topPane IS LIKE a TopPane.  
    topDispatcher IS LIKE a TopDispatcher.  
    model.      "the model controlling the pane/dispatcher"  
  
synonyms:  
    pane = dispatcher pane.    "the dispatcher's pane is the"  
                                "pane participant"  
    dispatcher = pane dispatcher. "the pane's dispatcher is"  
                                "the dispatcher participant"  
  
    topPane = pane topPane  
    model = pane model.  
.  
.
```

Figure 24: Section 2 of the Pane/Dispatcher group.

invariants:

```
(pane contents REFLECTS model contents) USING update.
    "the pane should reflect the data known by the model"
```

Figure 25: Section 3 of the Pane/Dispatcher group.

The third section of a group's declaration (see figure 25) specifies the *invariants*. Invariants are an important aspect of a group because, they identify information the group's members cooperate to maintain. In other words, invariants identify dependencies between the objects. When specifying an invariant it is necessary to be able to specify the objects maintaining the invariant, and the mechanism through which the invariant condition is satisfied. Without both pieces of information it is difficult to ascertain the purpose and behaviour of the invariant. PEGs identifies the invariant condition in high level terms, and specifies the path expression which causes its satisfaction. The form of an invariant is:

```
predicateCondition USING pathExpression
```

The Pane/Dispatcher invariant (see figure 25), demonstrates several aspects of the PEGs notation.

```
(pane contents REFLECTS model contents) USING update
```

The **reflects** operator, adopted from [Helm, et al. 90] specifies the existence of a relationship between the objects. The exact nature of the relationship is not relevant at this level of description. The above invariant declaration specifies the pane's contents can be made to reflect the model's contents through the use of the path whose name is *update*. In PEGs, a path name can be substituted wherever a path expression is expected. This is discussed in detail later in this section.

The fourth and final section of a group's declaration (see figure 26) specifies the group's behaviour in two parts:

- high level behaviour (*behaviour field*)
- detailed behaviour consisting of:
 - typical interaction patterns (*paths field*)
 - atypical interaction patterns (*exception paths field*)

Paths within PEGs are not intended to be a formal definition of all possible interactions; instead they are used to capture the essential and typical patterns of interaction.

The *behaviour* path expression field captures the high level view, with subsequent *named paths* providing a more detailed view of the interactions. The behaviour path expression usually contains references to other paths described in the *paths* field of the group's declaration. Consequently, PEGs not only supports the decomposition of the system into groups, but also supports the decomposition of the interactions within a group. This feature allows the client to quickly grasp the typical behaviour while at the same time providing specific details about the behaviour in the event they are required.

The behaviour path of the Pane/Dispatcher group (see figure 26) specifies the *create* path typically occurs first, followed by the *open* path. This is eventually followed by a sequence of window activations and deactivations with intervening updates and user interactions occurring repeatedly. Ultimately the Pane/Dispatcher group terminates.

The *paths* field contains detailed path expressions whose form is:

```

    pathName:    typicalInitiator >> pathExpression
or
    pathName:    pathExpression

```

The term *typicalInitiator >>* specifies the path is typically initiated by the object *typicalInitiator*. In the event several objects typically initiate the path, the following form is used:

```
(A | B) >>
```

This specifies A or B typically initiates the path. This notation is similar to that of the *IS LIKE* clause described earlier. To indicate paths typically initiated by a user action, the keyword *USER* is given as the typical initiator (see path *userInput*: of figure 27 on page 73). *pathName* is a brief label describing the path and hence the set of interactions the path describes. The above named paths have several advantages:

- The ability to specify the originator allows the identification of the “typical” case. As mentioned earlier, any object can potentially send a message to any other object, the aim is to capture the typical patterns that arise.
- Associating a descriptive label with a path expression has the effect of attaching meaning to the important sets of interactions exhibited by the group. This effectively does for interactions what Pernici’s roles do for states, that is, to partition the behaviour into meaningful named parts.
- Associating a label with a path allows the path to be referenced by other paths. This prevents unnecessary repetition of path expressions and improves readability.

As mentioned above, paths can reference other paths by referencing their name. In the Pane/Dispatcher example, *update* path typically occurs once the window has been activated (as indicated by the behaviour path). The path is typically initiated by the *model* and results in either the *refresh* path or one of the other three paths occurring. Closing the window (the *close* path), is typically initiated by the *topPane* (which *IS LIKE* a *TopPane*). The associated path shows the set of interactions include *close* being sent to *pane*; followed at some point by *release* being sent to the *model*.

The *exception paths* field contains detailed path expressions describing the interaction patterns associated with the occurrence of an exception. PEGs maintains exception paths separately in order to distinguish typical behaviour from atypical behaviour. The Pane/Dispatcher group does not define any exception paths.

behaviour:

```

create ;
open ;
(windowActivating;
  (update*;
    cursorEntersPane;
    (update | userInput)*;
    cursorLeavesPane
  )*;
  windowDeactivating
)*
terminate.

```

paths:

```

create:                                "path to create the group"
  model >>
    pane model;;                      "specifies the model"
    [pane name:];                     "specifies the pane's name"
    [pane paneMenuSelector:]. "specifies a message that can"
                                "be sent to the model to obtain the menu"

terminate:                             "path to terminate the group"
  close.

open:
  topPane >>
    pane reframe;; "determine rectangle for the pane's area"
    pane open.

close:
  topPane >>
    pane close ;
    model release. "break connections between the model"
                  "and the pane"

```

Figure 26: Section 4 of the Pane/Dispatcher group (part 1 of 2).

```

update:
  model >>  "update the displayed information"
    refresh
    | (pane update: ; refresh)
    | (pane update:with: ; pane perform:)
    | (pane update:with:with: ; pane perform:with:).
cursorLeavesPane:
  USER >>
    dispatcher processInput ; dispatcher deactivate .
cursorEntersPane:
  topDispatcher >>
    dispatcher processInput ; dispatcher activate .
windowActivating:
  topPane >>
    pane displayWindow .
windowDeactivating:
  topPane >>
    pane graySelection .
userInput:  "handle user input"
  USER >>
    dispatcher processInput ; dispatcher processKey ;
    (menuRequest | keyInput | ctrlKeyInput |
      functionKeyInput).
menuRequest:
  dispatcher processFunctionKey ;
  (topPane performMenu
    | (pane performMenu; (topPane performMenu
      | model perform:))).
functionKeyInput:
  (dispatcher select
    | (dispatcher processKey: ;
      dispatcher processFunctionKey:)).
.
.
exception paths:
  none

```

Figure 27: Section 4 of the Pane/Dispatcher group (part 2 of 2).

3.6.3 Inheritance

Groups can be arranged into inheritance hierarchies. Groups can also be designated as abstract groups. Abstraction and inheritance are essential elements in promoting reuse of groups. The *Pane/Dispatcher* group shown in figure 23 is an abstract group. Abstract groups are intended to be specialized much in the same manner as abstract classes are intended to be specialized. The *TopPane/TopDispatcher* group shown in figures 29-31 is a subgroup of the abstract *Pane/Dispatcher* group.

The *TopPane/TopDispatcher* group demonstrates several new aspects relating to fields:

- A group inherits all of its fields from its supergroup. For example, *TopPane/TopDispatcher* group inherits all of its fields from *PaneDispatcher*.
- Fields left blank, (eg. *uses* and *exceptions*) are not specialized by *TopPane/TopDispatcher*.
- A group specifies it is adding to the inherited field by using "with super and;" at the start of the field. For example, the *description* field of *TopPane/TopDispatcher* specifies that it consists of the *description* defined in the super group, along with the specified additional text. Similarly, the set of interaction paths for *TopPane/TopDispatcher* consists of the paths defined in *Pane/Dispatcher* along with those specified in *TopPane/TopDispatcher*. The "with super and;" construct is declarative, not executable.

The above mechanisms support the requirement, as outlined in Chapter 1, to provide for the inheritance of groups. In addition to the large grain inheritance of groups, PEGs also supports finer grain inheritance by allowing the specialization of individual fields of a group's specification. Consequently, these mechanisms allow subgroups to clearly identify the specific properties they refine.

```
create
model  >>
    super create &
    (pane addSubpane:*;
     [pane label:];
     [leftIcons:];
     [rightIcons:]
    );
```

Figure 28: The create path.

A group inherits the path expressions of its supergroup. Path expressions are not restricted to referencing paths defined in the same group; inherited paths can also be referenced. Section 3.6.4 examines mechanisms for referencing the path expressions of other groups. To reference a path defined within the group, or inherited by the group, one of two mechanisms is used:

1. *pathName*
2. *super pathName*

The first method specifies the path whose name is *pathName* should be used. PEGs examines the group to determine if it explicitly defines the path. If no path is found, PEGs examines the group's supergroup. This process is repeated up the inheritance hierarchy until a path by the given name is found. In the second method, the search process begins with the group's supergroup instead of the group.

TopPane/TopDispatcher's *create* path (see figure 28) specifies the sequence of interactions involved in creation includes those described in the *create* path of the supergroup and then the path in parenthesis or vice versa (by definition of the & operator). The above path could also have referenced the inherited *userInput* path.

The `TopPane/TopDispatcher` example illustrates another aspect of the PEGs notation intended primarily as a descriptive mechanism. The expression¹⁰,

`<.V. subpanes> operation1`

indicates the object receiving the message is actually a collection of objects and message `operation1` applies to members of the collection. The special notation is used to provide a more informative description than is possible with a generic description of the form `(a Pane operation1)*`.

PEGs fulfils the requirements outlined in Chapter 1 with respect to inheritance. We will now examine the composition of groups using as an example, a group that describes a sample window with two panes.

¹⁰The term `.V.` means "for all" and is used in place of $\forall$ which may not be typeable in some programming environments.

```
name:
    TopPaneDispatcher

descriptiveName:
    TopPane/TopDispatcher Framework

abstract:
    no

description:
    with super and;

    TopPane's control the areas of the window not occupied
    by any pane, this includes the label and its icons.

    TopPanes are responsible for ensuring the subpanes
    are all opened, closed, reframed and notified of
    window activations and deactivations

inheritsFrom:
    PaneDispatcher    "subgroup of PaneDispatcher group"

defines participants:
    with super and;          "add to inherited information"
    pane = a TopPane.
    dispatcher = a TopDispatcher .

uses:
    "no change to inherited field"

secondary participants:
    with super and;
    subpanes IS LIKE COLLECTION OF a SubPane.
```

Figure 29: A subset of the TopPane/TopDispatcher group.

synonyms

```
topPane = pane.  
topDispatcher = dispatcher.  
<.V. subpanes> toppane = pane .  
    "for all subpanes, their toppane"  
    "is the same as the groups pane"  
  
subpaneDispatchers = <.V. subpanes> dispatcher .
```

invariant:

```
with super and;  
(<.V. subpanes> frame REFLECTS topPane frame)  
    USING topPane reframe:.
```

behaviour:

```
create ;  
schedule ;  
(windowActivating;  
    (update*;  
        cursorEntersPane;  
            (update | userInput | collapse)*;  
            cursorLeavesPane  
        )*;  
    windowDeactivating  
)*  
terminate.
```

Figure 30: A subset of the TopPane/TopDispatcher group (cont. 2).


```

paths:
  with super and;      "add/refine the set of inherited paths"

  create
    model  >>
      super create & "reference the inherited path"
      ([pane label:] & "specifies the label"
       [leftIcons:] & "specifies icons for the label bar"
       [rightIcons:] &
       pane addSubpane:* "subpanes can be added"
      );

  open:
    model  >>
      pane reframe;;
      <.V. subpanes> reframe;;
      pane open;
      <.V. subpanes> open .

  close:
    (USER | model) >>
      (dispatcher closeWindow | dispatcher closeIt) ;
      <.V. subpanes> close;
      [model closingWindow];
      pane release.

  refresh:
    (pane | model) >>
      super refresh;
      model label;
      (Scheduler resume |
       (pane displayLabel; pane highlightLabel)) .
      .
      .

exception paths:

```

Figure 31: A subset of the TopPane/TopDispatcher group (cont. 3).

3.6.4 Composition

Groups can be composed of other groups. This results in an alternative hierarchy based on compositions where the high level groups use the more basic groups. The Sample Window group shown in figure 32, is a higher level group composed of other groups and a model participant. The Sample Window group describes a collection of objects and groups which interact to provide the user with a simple window consisting of two text areas, a fun text area and a boring text area. The user can save the text of either area. Saving the text in the fun area updates the text in the boring area.

Sample Window instantiates a TextPane/Dispatcher group twice along with a single TopPane/TopDispatcher group. The basic mechanism for using another group is

name = n OF groupName

where **n** is an integer indicating the number of occurrences of the group whose name is *groupName*. The term COLLECTION can be substituted for *n* to indicate a collection of occurrences of the specified group. *name* is the name to associate with the particular usage of the group. Sample Window contains three groups named, window, fun and boring.

The synonyms of Sample Window indicate the model participant corresponds to the model identified as a secondary participant of the other groups. The synonym expressions of Sample Window emphasize the point that groups are themselves objects and as such can be sent messages like any other object.

Sample Window also illustrates a more complex invariant composed of two invariants and the path expressions that satisfy them. The invariant expression.

```
(fun REFLECTS model funText)
    USING fun update.
& (boring REFLECTS model boringText)
    USING boring update.
```

specifies that the *fun* group reflects the model's fun text, and, the *boring* group reflects the model's boring text. This path shows how the *fun* and *boring* groups are treated as objects. The path expressions *fun update* and *boring update* are references to the *update* path expressions provided by their respective groups.

The behaviour path and its accompanying *windowCreate* path demonstrate the difference in perspective and the nature of abstraction that is possible. From the point of view of the Sample Window, the essence of the window is exhibited by its creation, the saving and updating and eventually its termination. The behaviour path captures the essential behaviour patterns and hides the specifics.

Examining the behaviour path, the interaction patterns exhibited by the group consist of creating and scheduling the window, followed by any number of window activations and corresponding deactivations. Any number of saves of the fun text, followed by the necessary update to the boring text, or saves to the boring text, can occur between activations and deactivations. The Sample Window behaviour path contains several references to the path expressions defined by other groups.

We have seen how PEGs allows a system to be divided into groups. We have shown how groups can be arranged into abstraction hierarchies or composed to form more complex groups, thus realizing a composition hierarchy where higher level groups build on more primitive groups.

We now examine the class and method specifications supported by PEGs.

```
name:
    SampleWindow

descriptiveName:
    Sample Window

description:
    A window with a toppane and two subpanes.

abstract:
    no

inheritsFrom:
    none

defines participants:
    model IS a SampleModel.

uses:
    window = 1 OF TopPaneDispatcher .
    fun = 1 OF TextPaneDispatcher.
    boring = 1 OF TextPaneDispatcher.

secondary participants:
    none

synonyms
    fun topPane = window pane .
    boring topPane = window pane.

    fun model = model.
    boring model = model.
    window model = model.
```

Figure 32: A subset of the Sample Window group.

```

invariants:
  (fun REFLECTS model funText) USING fun update.
  & (boring REFLECTS model boringText) USING boring update.

behaviour:
  windowCreate;
  window schedule;
  ((window windowActivating;
    ((fun saving;boring update) | boring saving)*
    "many saves and updates"
    window deactivatingWindow
  )*)
  [model closeSampleWindow];
  window closeIt .

paths:
  windowCreate
    model>>
      fun create;
      boring create;
      window create .

exception paths:
  none

```

Figure 33: A subset of the Sample Window group (cont. 2).

3.6.5 Describing Classes and Methods

PEGs supports the description of classes and methods. The aim in providing this finer granularity of description is to highlight certain aspects of classes and their methods, it is not intended to provide a detailed formal specification of each class as is done in Procol [Van Den Bos and Laffra 89] or Fresco [Wills 91], [Wills 92]. An overview of the class and method descriptions follows. Examples using the pane/dispatcher/model framework are given in the appendix.

The PEGs description associated with a class consists:

description

A textual description of the class.

abstract

Identifies if the class is an abstract class (see below).

variables

A collection of variables, and their typical type(s). These are specified in a manner similar to the *IS LIKE* clause of groups. For example in the expression.

dispatcher IS LIKE a TopDispatcher

specifies that the variable dispatcher is "typically" like a TopDispatcher. The motivation for *IS LIKE* clause for variables is identical to that for groups. The *LIKE* term can be omitted if a stronger assertion is desired.

As with groups, a class inherits the properties of each field from its superclass. Fields left blank indicate the class does not change the inherited information. The *super* mechanism can also be used within fields.

The PEGs description associated with a method contains:

description:

A textual description of the method.

abstract:

Identifies if the method is an abstract method (see below). In addition a method can be designated as providing default behaviour which should typically be specialized. It is important to distinguish methods which require specialization from those which provide a default behaviour or those which provide a complete implementation. The valid values for this field are yes, no and default

precondition:

A predicate condition specifying conditions which must be satisfied before the message is sent to the object. PEGs uses simple predicates to describe preconditions. Predicates supports the conjunction and disjunction operations and allows for predicate operations on objects.

postcondition:

A predicate condition specifying conditions that holds after the method has been performed. The form of the postcondition is similar to the precondition.

input:

An expression defining the input parameters and their expected types or values.

output:

An expression defining the output value for the method. For example, *true* | *false* specifies the method returns either true or false and *true* | *an Integer* specifies either true or an a kind of Integer is returned.

obligation:

A path expression describing any interactions the sender must satisfy at a later time.

Methods can specify a path expression to identify obligations the sender acquires as a result of requesting the service. For example, in invoking an open operation on a file the sender acquires the obligation to ensure the path *close* is performed. A path of *flush;close* indicates the sender must eventually invoke the flush operation, followed eventually by the close operation. PEGs' obligations are adopted from [Perry 87] and [Perry 89].

PEGs uses informal comments to define the conditions under which the values occur. This view stems from the belief that a brief description is quicker to grasp and more applicable at the granularity of PEGs than a formal notation which may potentially require mentally switching back and forth between a programming language and a formal language. The advantages and disadvantages of specifying preconditions and post conditions are discussed in [Meyer 90] who argues in favor of them. Meyer argues it is important for clients to know the constraints of an operation and be provided with the knowledge that the behaviour of the operation is guaranteed if the conditions are met.

Figure 34 contains an example of the `update:with:with:` method (typically sent by a model when a pane should refresh itself) and illustrates the regular expression style used in describing each of the fields. In figure 34 the three inputs are described with a single expression.


```
method:    update:with:with:
of class Pane

description:
    The data being presented may have changed.
    so update the contents of the pane if the
    specified name matches the pane's.

abstract:
    no

precondition:
    isOpen

postcondition:
    none

input:
    (nameSymbol IS a Symbol)
    & (aSymbol IS a Symbol)
    & (anObject IS an Object)

output:
    none

obligation:
    none
```

Figure 34: Method specification

3.6.6 Equivalences

Path expressions in PEGs support several expressive forms not found in the traditional path expression notation as described by Bruegge and Hibbard. These additions are provided for convenience and to help clarify the expected behaviour. They are syntactic sugaring for underlying traditional path expressions. Following are the PEGs expressions and their equivalent traditional notation:

<code>A+</code>	Is equivalent to <code>A;A*</code> .
<code>[A]</code>	Is equivalent to either A or nothing. For example, <code>[A];B</code> is equivalent to <code>(A;B) B</code> .
<code>A & B</code>	Is equivalent to <code>(A ; B) (B ; A)</code> .
<code><.V. subpanes> operation</code>	Is equivalent to <code>(a SubPane msg)*</code> . The "for all" subpanes notation acts purely to clarify the interaction trace.

Referencing one path from another is equivalent to replacing the reference with the entire path being referenced. For example,

```

A msg1; path1; B msg2
where
  path1: C msg3; D msg4

```

is equivalent to replacing the referenced path, `path1`, with its path expressions to yield:

```

A msg1; C msg3; D msg4; B msg2

```

2

of/de

2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

PIONEERS IN METHYLENE BLUE TESTING SINCE 1974

1480 COUNTY ROAD 5, RUMBLEVILLE, MINN 55357 USA
TEL: 612/435-7957 FAX: 612/435-7957 TEL: 914/895-0888

3.7 PEGs Syntax

We now present an extended BNF for the PEGs notation:

Operation	Meaning
{A}	A is repeated zero or more times.
[A]	A is option.
"..."	The text between the double quotes must appear in its entirety.
A B	Either A or B occurs.
()	Expressions between parenthesis are treated as a single item.

3.7.1 Groups

The BNF for path expression groups is:

```
group =
  "name:" groupName
  "descriptiveName:" text
  abstract
  description
  "inheritsFrom: (" none | groupName)
  "defines participants:"
    [none |
    ([super]
    {name "IS" prefix className end}
    )]
  "uses:" [none |
    ([super]
    {name "=" (number | "COLLECTION") "OF" groupName end}
    )]
  "secondary participants:"
    [none |
    ([super]
    {name
      [defineType]
      [SUPPORTING {message ";"}] end}
    )]
```

```

"synonyms:"
    [none | ([super]
        {(object [message] = object [message] end)}})]

"invariant:"
    [none | ([super] {invariantList} end)]

"behaviour:" [none | pathExpression]

"paths:"
    [none | ([super] | {pathDefinition end})]

"exception paths:"
    [none | ([super] | {pathDefinition end})]

abstract = "abstract: ["yes" | "no" ]
description = "description:" [none | ([super] text)]

target = object message
object = name | all
all = "<.V." name ">"
andor = "&" | "|"
prefix = ["a" | "an"]
none = "none"
super = "with super and;"
message = name | selector

defineType =
    (((("IS" ["LIKE"] ["COLLECTION OF"]) prefix className )
        | ( defineType "|" defineType)

pathDefinition =
    [[object] | "USER" ">>"] pathExpression

invariantList = "(" invariant ")" "USING" path end
                {andor "(" invariantList ")" }

invariant = (target "REFLECTS" target)
            | "(" invariant ")"
            | (invariant andOr invariant)

```

```

path = pathReference | pathExpression
pathReference = [object | "super" | "self"] name
pathExpression =
    ("(" pathExpression ")")
    | (pathExpression ("*" | "+"))
    | (pathExpression (";" | "&") pathExpression)
    | ("[" pathExpression "]")

groupName = className
name = smallLetter {letter | digit}
className = capitalLetter {letter | digit}
text = {character}
character = {digit | letter | special}
number = {digit}
letter = capitalLetter | smallLetter

capitalLetter = "A" | "B" | "C" | "D" | "E" | "F" | "G" |
    "H" | "I" | "J" | "K" | "L" | "M" | "N" |
    "O" | "P" | "Q" | "R" | "S" | "T" | "U" |
    "V" | "W" | "X" | "Y" | "Z"

smallLetter = "a" | "b" | "c" | "d" | "e" | "f" | "g" |
    "h" | "i" | "j" | "k" | "l" | "m" | "n" |
    "o" | "p" | "q" | "r" | "s" | "t" | "u" |
    "v" | "w" | "x" | "y" | "z"

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
    "8" | "9"

end = "."

special = "~" | "!" | "0" | "@" | "$" | "%" | "^" | "&" |
    "*" | "(" | ")" | "_" | "+" | "<" | "-" | "=" |
    "{" | "}" | "|" | "[" | "]" | "\" | ":" | ";" |
    ">" | "<" | ">" | "?" | "," | "." | "/"

```

selector = a Smalltalk symbol as defined on page 493 of [Digitalk 88].

3.7.2 Class

The BNF for a class specification is shown below. The BNF references some terms defined in the BNF for groups.

```
class =
  description
  abstract
  "variables:"
    [none |
      ([super] {name [defineType] end})]
```

3.7.3 Method

The BNF for a class specification is shown below. The BNF references some terms defined in the BNF for groups.

```
method =
  description
  "abstract:" ["yes" | "no" | "default"]
  "precondition:" [none | predicate]
  "postcondition:" [none | predicate]
  "input:" [none | input]
  "output:" [none | output]
  "obligation:" [none | pathExpression]

predicate =
  (predicate andor predicate)
  | (([object] message) | "super")
  | ("(" predicate ")")
  | (target REFLECTS target)
  | ("~" predicate)

input =
  ((name defineType) | super | object message)
  | ("(" input ")")
  | (input "|" input)
  | (input "&" input)

output =
  | ("(" output ")")
```

```

| (object | literal | object message | super
  | ([prefix] className) )
| (output "|" output)
| (output "&" output)

```

literal = a Smalltalk literal as define on page 493 of [Digitalk 88]

3.7.4 BNF Summary

The above sections presented the BNF for the different elements of PEGs. The precedence for operators used within PEGs is (in order of decreasing precedence)

- ()
- []
- ~
- reflects
- & |
- + *
- =
- ;

Comment in PEGs can appear at any location. A comment is defined as:

```
comment = "" {"" | character} ""
```

3.8 Summary

This chapter presented the PEGs notation. PEGs allows the organization of the system into groups of interacting objects. Groups can be arranged into abstraction hierarchies or composed to form more complex groups, thus realizing a composition

hierarchy where higher level groups build on more primitive groups. Interactions within a group are described using a path expression notation based on the work of Campbell and Habermann [Campbell and Habermann 73] and Bruegge and Hibbard [Bruegge and Hibbard 83]. The path expression notation is based on regular expressions. A path defines a set of possible traces, or more formally, a path expression can be viewed as a grammatical object generating a trace language. A trace is the set of interactions resulting from a specific execution of the system. For this reason, path expressions are a mechanism for describing the sets of executions that typically occur. Groups are first class objects, unlike [Helm, et al. 90] in which they are viewed as a conceptual entity.

The use of path expressions allows PEGs to build on the research already conducted into the area of path expressions. Moreover, the rules for constructing path expressions are simple and well known. PEGs extends the traditional path expression notation with additional operators. It was shown that these new mechanisms can be mapped to the traditional path expression notation. Path expressions focus attention on the critical interactions. PEGs is useful in describing frameworks for several reasons. First, it allows a framework to be described in terms of its dynamic behaviour, notations of the previous chapter focussed on the behavior within a single object. Second, it describes groups, objects and methods using a collection of fields. Fields can be independently specialized by subgroups, classes or methods, respectively. This approach makes the changes introduced by in the sub group/class/method explicit. PEGs also allows groups/classes/methods to be identified as either abstract or concrete and in the case of methods whether the behaviour is simply a default behaviour. This helps identify aspects of the framework which are either incomplete or provide a default behaviour and hence need to be specialized. As Helm et al. [Helm, et al. 90] observe, specializing an object that interacts closely with other objects often involves the specialization of several participants. The group mechanism allows the group as a whole to be refined thus

	Schlaer and Mellor	Rumbaugh et al.	Bear et al.	Wirfs- Brock et al.	Pernici	Helm et al.	PEGs
diagram based	yes	yes	yes	yes	no	no	no
design methodology	yes	yes	no	yes	no	no	no
groups	yes	no	no	no	no	yes	yes
group inheritance	no	no	no	no	no	yes	yes
group composition	yes	no	no	yes	no	yes	yes
event traces	yes	yes	yes	no	no	no	yes (defined by paths)
object description	state diagram	harel diagram	harel diagram	collab- orations	roles	formal operat- ions	yes (class and method specific- ations)
static interaction picture	yes	no	yes	yes	no	no	no
dynamic of interactions	no	no	no	no	no	no	yes
formal ¹¹ specification	no	no	no	no	no	yes	no
identifies invariants	no	no	yes	no	yes	yes	no

Figure 35: Summary of notations with PEGs.

localizing and clearly identifying the set of required specializations. Finally, PEGs places the emphasis on the reuse of groups of objects and patterns of interactions instead of on the individual object. Figure 35 summarizes the notations of Chapter 2 and the PEGs notation.

The next chapter summarizes the thesis and discusses potential future research. The examples used in this chapter appear in their entirety in the appendix.

¹¹Although state diagrams like Harel's can provide a formal description, the above notations do not use them for that purpose. The diagrams are used to capture analysis and design information and are insufficient to provide a formal specification of the system.

Chapter 4: Summary

4 Conclusion

This chapter summarizes the PEGs notation and its contribution. Directions for future research are discussed including the types of tools that could be used to assist the developer when working with an object-oriented system described using PEGs.

We have presented the PEGs notation, a method for describing the behaviour of interacting objects within an object-oriented system. Sets of interacting objects are formed into groups called path expression groups (PEGs). The behaviour of a group is described using path expressions, a concise method of specifying the patterns of interaction exhibited by the group.

In Chapter 2, we examined some of the current notations for describing a system of objects. In general, these notations focus on the individual object and its internal behavior. Description of object interactions is limited to a static snapshot capturing the system at an instant in time when every interaction simultaneously occurs – a somewhat unrealistic viewpoint. To understand a system of objects it is necessary to understand the manner in which they interact and cooperate to perform a set of tasks. Objects do not exist in isolation, consequently notations based solely on an isolationist view are inherently insufficient.

PEGs incorporates a group mechanism similar to that described by Helm et al. In [Helm, et al. 90], groups of objects were modeled as contracts. PEGs, like Helm et al. supports the inheritance and composition of these groups. Unfortunately, Helm et al. do not capture the interactions within or among groups. Their notation focuses on the behaviour of an object's operations and the causal obligations of

objects. Other notations do not treat groups as first class entities and fail to support their inheritance and composition.

PEGs describes the interactions within a group, and between groups, using path expressions, a notation developed by Campbell and Habermann [Campbell and Habermann 73]. Path expressions have previously been used within the area of concurrency to define the valid orderings of operations on a set of objects. Chapter 3 described path expressions and illustrated their use. PEGs differs from the earlier applications of path expressions in several ways:

- PEGs applies path expressions to groups of objects rather than at the operation level.
- PEGs uses path expressions to capture the **typical** behaviour of a system. Previous applications of path expressions were concerned with process synchronization and therefore required to define the set of all valid orderings of operations on a given object. PEGs' approach is motivated by the observations of [Lalonde, et al. 91], [Lalonde 92] and, [Schlaer and Mellor 92] who note that although an object can potentially communicate with any other object, typically only a few important patterns are observed.
- PEGs' path expressions support inheritance. A group inherits the path expressions of its supergroup. Path expressions can reference inherited paths in a manner analogous to the "super" mechanism of Smalltalk methods. In addition, PEGs paths are named; this in turn permits composition of paths. A path can reference paths defined by the group, inherited from the supergroup or, if the group uses other groups, a path provided by one of the groups.

In addition to path inheritance, specifications associated with groups, classes and methods support inheritance. A subgroup, subclass or method, can refine individual fields (eg. description) of an inherited specification. This mechanism supports the indication of the particular aspects of the behaviour that is being specialized.

PEGs is useful in describing frameworks (as defined by [Wirfs-Brock, et al. 90b]), in that it provides a mechanism for specifying the dynamic behaviour of a framework. Frameworks can be organized into groups. Specialization of framework classes occurs through the specialization or composition of groups. This in turn allows changes made in the process of using a framework to be isolated within a group. Groups, classes and methods can be designated as providing abstract behaviour. Thus, PEGs helps identifying both the overall behaviour and the aspects of the framework requiring specialization. Lastly, through the inheritance mechanisms provided for specifications, clients of a framework can identify changes introduced.

4.1 Future Research

The PEGS notation has brought out some interesting ideas relating to the description of systems of objects. This thesis has presented the initial efforts on the PEGs notation and several issues remain to be investigated:

- PEGs uses the basic path notation and does not include constructs to support predicate path expressions. It is uncertain whether predicates would significantly contribute to the high level description of a system's behaviour, or whether they would complicate the description without considerable gain. The need for predicate path expressions can only be determined by describing a larger number of systems.
- PEGs describes a system in terms of the behaviour it typically exhibits. An open question is whether or not it is beneficial to use PEGs to provide a formal description of the system. One possibility is to use a hybrid of PEGs and another notation supporting a more formal description. This approach would have the benefit of using each notation for its intended purpose while allowing the system to be viewed in multiple ways.

- PEGs does not require specific tools; however, the development of certain types of tools would be beneficial. These include:
 - Tools supporting a message-oriented view: Traditional tools force users into a class-oriented view. Additional support for a message-oriented view would allow clients to view the system from different perspectives.
 - Tools for flattening a specification hierarchy: Clients viewing an object are often uninterested in whether the behaviour is provided by the object or inherited. PEGs provides a super mechanism for fields within a specification. Tools can be constructed to provide either the traditional view, or a client view in which references to super are replaced with the inherited information yielding a flat view of the group, class or method.
 - Tools to propagate changes to path expressions affected by a modification. The need for propagation tools is common to all notations.
- PEGs is a text based notation. It is sometimes helpful to be able to view the system pictorially. Path expressions can be visualized in one of two ways:
 1. by visualizing the behaviour described by the path expression
 2. by visualizing the structure of the path expression

The first approach is currently an open question with respect to PEGs. Viewing the structure of a path expression can be accomplished by converting the path to a finite state machine. Unfortunately, such an approach may not be ideal. As Zave [Zave 85] observes, the primary problem with finite state machines is the difficulty human readers have in comprehending them. An alternative is to use Zave's sequence diagrams which use a tree structure to break up a sequence into subsequences. Sequence diagrams are equivalent to regular expressions [Zave 85]. Path expressions are equivalent to regular ex-

pressions, therefore path expressions can be represented as sequence diagrams without the loss of information.

A problem encountered with visualizing a system arises when trying to automate the process. It is generally a difficult problem to generate what Harel [Harel 92, pp. 13] terms "nice looking" diagrams. Zave's sequence diagrams are tree-based and consequently tend not to suffer from layout problems. In considering notations for visualizing the behaviour described by the path expressions, it is anticipated this problem will play a more significant role.

- Lalonde [Lalonde 92] proposes the development of tools for the regeneration of an object-oriented system's specification from its implementation. Lalonde asserts the need for regeneration tools is essential when faced with a rapid prototyping approach to development and development strategies which do not use a traditional waterfall model. A potential starting point for the regeneration of a system is sets of event traces produced by the system. It is feasible that tools could be used to generate path expressions (from these traces) as a step towards a comprehensive description of the system.
- In this, PEGs was presented within the domain of traditional sequential object-oriented systems. The potential applications of PEGs to real-time and concurrent systems requires investigation.
- Bruegge and Hibbard [Bruegge and Hibbard 83] describe a path expression mechanism for debugging. In their approach, a path expression specifies a sequence of operations that occur prior to the triggering of a debugger action (eg. breakpoint). This approach is more powerful than traditional approaches where the trigger typically occurs after a single operation or repetition of a given operation. A path expression mechanism allows arbitrary sequences to trigger the debugger actions. PEGs can similarly be used as a debugging

mechanism. A PEGs-based debugger would trigger debugger actions after arbitrary sequences of object interactions.

- Another possible application to debugging is to compare an event trace for a particular execution of a system against the path expressions describing the system. If the trace cannot be generated from the path expressions then either:
 - The path expressions are incorrect which may in turn imply the system is not sufficiently understood.
 - or,
 - A problem exists with the implementation.

The application of PEGs to testing and debugging needs to be examined by way of a prototype PEGs based debugger.

- Janicki[Janicki 85] describes an equivalence notion for path expressions. The extent to which these equivalence notions can be incorporated into PEGs is unknown. Minimally, it is anticipated that PEGs can be used to determine whether two groups are interaction-equivalent. That is, whether they exhibit the same patterns of interaction. In general however, comprehensive equivalence of programs is a difficult problem.

4.2 The Final Word

PEGs provides one view of a system of objects. Understanding and developing object-oriented systems often requires the development environment to simultaneously support several different views. Static views depicting the structure of objects and their relationships to each other at a single moment in time are a useful starting point when trying to understand a system [Rumbaugh, et al. 91, pp. 84]. The combination of a static view, such as one of those presented in Chapter 2, with

a PEGs dynamic view can provide developers with the necessary tools to better understand their system.

5 References

References

- [Andler 79] Andler, S., *Predicate Path Expressions: A High-Level Synchronization Mechanism*, Ph.D. Thesis, Carnegie Mellon University, 1979.
- [Barret, et al. 79] Barrett, W. et al., *Compiler Construction Theory and Practice*, Science Research Associates Inc., 1979.
- [Bear, et al. 90] Bear, S. et al., *Graphical Specification of Object Oriented Systems*, ECOOP/OOPSLA'90 Conference Proceedings, Ottawa, Canada, October, 1990, pp. 28-37.
- [Beck and Cunningham 89] Beck, K., Cunningham, W., *A Laboratory for Teaching Object-Oriented Thinking*, OOPSLA'89 Conference Proceedings, New Orleans, Louisiana, October 1989, pp. 1-10.
- [Brooks 87] Brooks, F., *No Silver Bullet: Essence and Accidents of Software Engineering*, IEEE Computer Vol. 20, No. 4, April 1987, pp. 10-19.
- [Bruegge and Hibbard 83] Bruegge, B. and Hibbard, P., *Generalized Path Expressions: A High-Level Debugging Mechanism*, Journal of Systems and Software, Vol. 3, 1983, pp. 265-276.
- [Campbell and Habermann 73] Campbell, R. and Habermann, A., *The Specification of Process Synchronization by Path Expressions*, Lecture Notes in Computer Science, Vol. 16, Springer Verlag, 1973, pp. 89-102.
- [Digitalk 88] Digitalk, *Smalltalk/V286, Object-Oriented Programming System. Tutorial and Programming Handbook*, Digitalk Inc., 1988. Smalltalk/V286 is a registered trademark of Digitalk Inc.
- [Cockburn, et al. 91] Cockburn, A. et al., *Interactive Prototyping of Communication-Intensive Systems*. First International Workshop on Rapid System Prototyping, Research Triangle Park, North Carolina, June 1990.
- [Cook 92] Cook, W., *Interfaces and Specifications for the Smalltalk-80 Collection Classes*, To be published in OOPSLA'92 Conference Proceedings, Vancouver, British Columbia, 1992.
- [Gibbs, et al. 90] Gibbs S. et al., *Class Management for Software Communities*, Communications of the ACM, Vol. 33 No. 9, September 1990, pp. 90-103.
- [Goldberg and Robson 85] Goldberg, A. and Robson, D., *Smalltalk-80 The Language and its Implementation*, Addison-Wesley, 1985.

- [Goldsack and Moreton 82] Goldsack, S. and Moreton, T., *Ada package specifications: path expressions and monitors*, IEE Proceedings Vol. 129, No. 9, March 1982, pp. 49-54.
- [Harel 87] Harel, D., *On Visual Formalisms*, Technical Report, CMU-CS-87-126, Carnegie-Mellon University, Pittsburgh.
- [Harel 92] Harel, D., *Biting the Silver Bullet*, IEEE Computer, January 1992, pp. 8-20.
- [Helm, et al. 90] Helm, R., Holland, I. and Gangopadhyay, D., *Contracts: Specifying Behavioral Compositions in Object-Oriented Systems*, ECOOP/OOPSLA'90 Conference Proceedings, Ottawa, Canada, October, 1990, pp. 169-180.
- [Holland 92] Holland, I., *Specifying reusable components using Contracts*, Personal communication of work in progress, College of Computer Science, Northeastern University, 1992. To be published in ECOOP'92.
- [Janicki 85] Janicki, R., *An Equivalence Notion for Path Expressions Systems* Journal of Information Processing and Cybernetics, 1985, pp. 283-295.
- [Kalashnikova and Pankrats 91] Kalashnikova, V. and Pankrats, E., *Algorithm to Construct the Behavior of Concurrent Systems Represented by Modified Path Expressions*, Translated from Kibernetika No. 1, January 1991, pp. 35-39.
- [Lalonde, et al. 91] Lalonde, W., White, P. and McGuire, K., *Coordinators: A Mechanism for Monitoring and Controlling Interactions Between Groups of Objects*. Technical Report, School of Computer Science, Carleton University, April 1991.
- [Lalonde 92] Lalonde, W., *(Proposal) A Design Regeneration Tool for Object-Oriented Systems*, Personal communication of project proposal, Carleton University, 1992.
- [Lauer and Campbell 75] Lauer, P., Campbell, R., *Formal semantics for a class of high level primitives for coordinating concurrent processes*, Acta Information 5, 1975, pp. 247-332.
- [Madany, et al. 89] Madany, P. et al., *A Class Hierarchy for Building Stream-Oriented File Systems*, ECOOP'89 Conference Proceedings, July, Nottingham UK, pp. 311-328.
- [Meyer 90] Meyer, B., *Tools for the New Culture: Lessons from the Design of the Eiffel Libraries*, Communications of the ACM, Vol. 33 No. 9, September 1990, pp. 69-88.

- [Oldehoeft and Jennings 84] Oldehoeft, A. and Jennings, S., *Dataflow Resource Managers and Their Synthesis from Open Path Expressions*, IEEE Transactions on Software Engineering, Vol. SE-10, No. 3, May 1984.
- [Pernici 89] Pernici, B., *Objects with Roles*, Object-Oriented Development, Université de Genève, 1989, pp. 75-100.
- [Perry 87] Perry, D., *Software Interconnection Models*, ACM, 1987.
- [Perry 89] Perry, D., *The Inscape Environment*, ACM, 1989.
- [Rumbaugh, et al. 91] Rumbaugh, J. et al., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
- [Russo and Campbell 88] Russo, V. and Campbell, R., *Process Scheduling in Multiprocessor Operating Systems using Class Hierarchical Design*, OOPSLA'88 Conference Proceedings, San Diego, California, September 1988, pp. 248-257.
- [Russo and Campbell 89] Russo, V. and Campbell, R., *Memory and Backing Storage Management in Multiprocessor Operating Systems using Class Hierarchical Design*, OOPSLA'89 Conference Proceedings, New Orleans, Louisiana, October 1989, pp. 267-278.
- [Schlaer and Mellor 92] Schlaer, S. and Mellor, S. *Object Lifecycles Modeling the World in States*, Yourdon Press, 1992.
- [Shields 79] Shields, M., *Adequate Path Expressions*, Proceedings of the Symposium on the Semantics of Concurrent Computation, Evian-les-Bains, July, 1979, Lecture Notes in Computer Science, Vol. 70, Springer Verlag, 1979, pp. 249-265.
- [Skuce 88] Skuce, D., *Conceptually Oriented Design: A synthesis of objects, logic and natural language for software production*, Technical Report TR-88-34, Ottawa University, 1988.
- [Thomson 92] Thomson, D., *Specifying Smalltalk Component Interfaces*, Graduate Directed Studies, School Of Computer Science, Carleton University, May, 1992.
- [Van Den Bos and Laffra 89] Van Den Bos, J. and Laffra, C., *PROCOL: A Parallel Object Language with Protocols*, OOPSLA'89 Conference Proceedings, New Orleans, Louisiana, October 1989, pp. 95-102.
- [Wegner 87] Wegner, P., *Dimensions of Object-Based Language Design*, OOPSLA'87 Conference Proceedings, Orlando, Florida, October 1987, pp. 168-182.
- [Wills 91] Wills, A., *Capsules and types in Fresco*, ECOOP'91 Conference Proceedings, Geneva, Switzerland, July 1991, pp. 59-76.

- [Wills 92] Wills, A.. *Specification in Fresco*, Personal communication of work in progress, University of Manchester, 1992.
- [Wirfs-Brock and Johnson 90a] Wirfs-Brock, R. and Johnson R., *Surveying Current Research in Object-Oriented Design*, Communications of the ACM, Vol. 33 No. 9, September 1990, pp. 104-141.
- [Wirfs-Brock, et al. 90b] Wirfs-Brock R., Wilkerson, B. and Wiener, L., *Designing Object-Oriented Software*, Prentice Hall, 1990.
- [Zave 85] Zave, P., *A Distributed Alternative to Finite-State-Machine Specifications*, ACM Transactions on Programming Languages and Systems. Vol. 7, No. 1, January 1985, pp. 10-36.
- [Zweig and Johnson 90] Zweig, J. and Johnson, R., *Conduits: A Communication Abstraction in C++*, To be published in USENIX C++ Conference, 1990. As referenced in [Wirfs-Brock and Johnson 90a].

A Appendix A: Pane/Dispatcher Group

In this section we present the Pane/Dispatcher group. Definitions for the group and its elements are provided in the following order:

- The Pane/Dispatcher group.
- The NewPane class and its methods.
- The NewDispatcher class and its methods.

A.1 PaneDispatcher Group

```

name:
  PaneDispatcher
descriptiveName:
  Pane/Dispatcher Framework
description:
  A Pane and Dispatcher cooperate to control a particular region
  of a window. Dispatchers are responsible for processing I/O.
  Panes are responsible for presenting the data to the user.
  Panes communicate with a controlling model from which their data
  is obtained.
abstract:
  yes
inheritsFrom:
  none
defines participants:
  dispatcher IS a Dispatcher.
  pane IS a Pane.
uses:
  none
secondary participants:
  scheduler IS LIKE a DispatcherManager.
  topPane IS LIKE a TopPane.
  topDispatcher IS LIKE a TopDispatcher.
  model.
synonyms:
  pane = dispatcher pane.
  dispatcher = pane dispatcher.
  pane topPane = topPane.
  pane model = model.
invariants:
  (pane contents REFLECTS model contents) USING update.

behaviour:
  create ;
  open ;
  (windowActivating;
    (update*;
      cursorEntersPane;
      (update | userInput)*;
      cursorLeavesPane
    )*;
    windowDeactivating ;
  )*
  terminate.
paths:

  create:
    model >>

```

```

pane model::
  {pane name:};
  {pane paneMenuSelector:}.

terminate:
  close.

open:
  topPane >>
    pane reframe::;
    pane open.

close:
  topPane >>
    pane close ;
    model release.

update:
  model >>
    refresh
    | (pane update: ; refresh)
    | (pane update:with: ; pane perform:)
    | (pane update:with:with: ; pane perform:with:).

userInput:
  USER >>
    dispatcher processInput ; dispatcher processKey ;
    (menuRequest | keyInput | ctrlKeyInput | functionKeyInput).

menuRequest:
  dispatcher processFunctionKey ;
  (topPane performMenu
   | (pane performMenu; (topPane performMenu | model perform:))
  ).

keyInput:
  dispatcher processInputKey: .

ctrlKeyInput:
  dispatcher processControlKey: .

functionKeyInput:
  (dispatcher select
   | (dispatcher processKey: ; dispatcher processFunctionKey:)
  ) .

cursorLeavesPane:
  USER >>
    dispatcher processInput ; dispatcher deactivate .

cursorEntersRegion:
  topDispatcher >>
    dispatcher processInput ; dispatcher activate .

refresh
  (pane | model) >> pane update.

windowActivating:
  topPane >>
    pane displayWindow .

windowDeactivating:
  topPane >>
    pane graySelection .

exception paths:

```

none

A.2 NewPane Class

description:

Panes are responsible for showing the contents of data maintained by their model. A Pane's input is handled through its dispatcher.

abstract:

yes

variables:

superpane IS a TopPane .
 dispatcher IS a Dispatcher .
 model IS an Object .
 frame IS a Rectangle .
 name IS a Symbol .
 paneMenuSelector IS a Symbol .
 paneScanner IS a Symbol .
 curFont IS a Font .

A.3 NewPane Methods

METHOD activatePane

description:

Make the receiver active to indicate that the cursor is in the pane and hence the pane now has control.

abstract:

no

precondition:

none

postcondition:

dispatcher active

input:

none

output:

none

obligation:

deactivatePane
 | dispatcher deactivate

METHOD activateWindow

description:

The window (top pane) is activating, change any visual clues for the pane.

abstract:

yes

precondition:

topDispatcher active

postcondition:

none

input:

none

output:

none

obligation:

deactivateWindow

METHOD border

description:

Draw a border around the pane's frame

abstract:

no

precondition:
 dispatcher active
postcondition:
 none
input:
 none
output:
 none
obligation:
 none

METHOD close

description:
 Close the receiver and release any
 dependencies from the model.
abstract:
 no
precondition:
 isOpen
postcondition:
 ¬isOpen
input:
 none
output:
 none
obligation:
 none

METHOD deactivatePane

description:
 Make the receiver inactive to indicate that
 the cursor is not in the pane and hence the
 pane no longer has control.
abstract:
 no
precondition:
 active & ¬hasCursor
postcondition:
 ¬active
input:
 none
output:
 none
obligation:
 none

METHOD defaultDispatcherClass

description:
 Answer the class of the default dispatcher
 associated with the receiver.
abstract:
 yes
precondition:
 none
postcondition:
 none
input:
 none
output:
 Dispatcher
obligation:
 none

METHOD dispatcher

description:

Answer the receiver's dispatcher.

```
abstract:
  no
precondition:
  none
postcondition:
  none
input:
  none
output:
  a Dispatcher
obligation:
  none
```

METHOD dispatcher:

description:
Set the receiver's dispatcher to be
aDispatcher.

```
abstract:
  no
precondition:
  none
postcondition:
  none
input:
  aDispatcher IS a Dispatcher
output:
  none
obligation:
  none
```

METHOD displayWindow

description:
Display the contents of the pane and show any
visual clues (eg. selection).

```
abstract:
  yes
precondition:
  isOpen
postcondition:
  none
input:
  none
output:
  none
obligation:
  none
```

METHOD frame

description:
Answer the receiver's frame, answer
nil if the receiver does not have a frame yet

```
abstract:
  no
precondition:
  none
postcondition:
  none
input:
  none
output:
  a Rectangle
obligation:
  none
```

```

METHOD frame:
description:
    Set the receiver's frame to be aRectangle.
abstract:
    no
precondition:
    ~isOpen
postcondition:
    none
input:
    aRectangle IS a Rectangle
output:
    none
obligation:
    none

METHOD graySelection
description:
    The topPane is deactivating the windows so
    change the visual clues and gray any selection.

abstract:
    yes
precondition:
    isOpen
postcondition:
    none
input:
    none
output:
    none
obligation:
    none

METHOD hasCursor
description:
    Answer true if the cursor is in the receiver's
    region otherwise answer false.
abstract:
    no
precondition:
    none
postcondition:
    none
input:
    none
output:
    true      "cursor is in pane's frame"
    | false   "cursor is outside pane"
obligation:
    none

METHOD homeCursor
description:
    Home the cursor to the top left corner of the
    receiver's frame.
abstract:
    no
precondition:
    active
postcondition:
    none
input:
    none

```

```

output:
  none
obligation:
  none

METHOD menu
description:
  Answer the receiver's menu selector. This is
  the selector it sends to the model to obtain the menu.
  Answer nil if the model does not supply a menu
abstract:
  no
precondition:
  none
postcondition:
  none
input:
  none
output:
  nil          "model does not supply method to obtain menu"
  | a Symbol
obligation:
  none

METHOD menu:
description:
  Set the receiver's menu selector. This is
  the selector it sends to the model to obtain the menu.
  Nil indicates the model does not supply a menu.
abstract:
  no
precondition:
  none
postcondition:
  none
input:
  (aSymbol isNil) | (aSymbol IS a Symbol)
output:
  none
obligation:
  none

METHOD model
description:
  Answer the model controlling the pane/dispatcher pair.
abstract:
  no
precondition:
  none
postcondition:
  none
input:
  none
output:
  an Object
obligation:
  none

METHOD model:
description:
  Set the model controlling the pane/dispatcher pair.
abstract:
  no
precondition:
  none

```

postcondition:

none

input:

anObject IS an Object

output:

none

obligation:

release | close

METHOD name

description:

Answer the name of the receiver.

abstract:

no

precondition:

none

postcondition:

none

input:

none

output:

a Symbol

obligation:

none

METHOD name

description:

Make the receiver active to indicate that
the cursor is in the associated pane and hence the
pane now has control.

abstract:

no

precondition:

isControlWanted

postcondition:

active

input:

none

output:

none

obligation:

deactivate

METHOD open

description:

Open the receiver

abstract:

yes

precondition:

"isOpen

& "active

& name notNil

& model notNil

postcondition:

isOpen

input:

none

output:

none

obligation:

close

METHOD paneScanner

description:

Answer the drawing device used for drawing the contents

of the pane.

```
abstract:
  no
precondition:
  none
postcondition:
  none
input:
  none
output:
  a CharacterScanner
obligation:
  none
```

METHOD performMenu

description:
Obtain the menu and pop it up
at the cursor location.

```
abstract:
  no
precondition:
  active
postcondition:
  none
input:
  none
output:
  a Symbol      "the selector corresponding to the chosen entry"
  | nil         "nothing selected from the menu"
obligation:
  none
```

METHOD popUp:

description:
Pop up the menu at the cursor location.

```
abstract:
  no
precondition:
  active
postcondition:
  none
input:
  aMenu IS a Menu
output:
  a Symbol      "the selector corresponding to the chosen entry"
  | nil         "nothing selected from the menu"
obligation:
  none
```

METHOD popUp:at:

description:
Pop up the menu at the the given location.

```
abstract:
  no
precondition:
  active
postcondition:
  none
input:
  (aPoint IS a Point)
  & (aMenu IS a Menu)
output:
  a Symbol      "the selector corresponding to the chosen entry"
  | nil         "nothing selected from the menu"
obligation:
```

```

    none

METHOD  reframe.
description:
    Compute the receiver's frame given aRectangle as
    the frame of the entire window (topPane).
    Adjust any drawing tools to match the new frame.
abstract:
    yes
precondition:
    none
postcondition:
    none
input:
    aRectangle IS a Rectangle
output:
    none
obligation:
    none

METHOD  release
description:
    Inform the model to release any dependencies
    it has on the receiver.
abstract:
    no
precondition:
    model notNil
postcondition:
    none
input:
    none
output:
    none
obligation:
    none

METHOD  showSelection
description:
    Change the visual cue of the selection
    to reflect a activated window.
abstract:
    yes
precondition:
    isOpen
postcondition:
    none
input:
    none
output:
    none
obligation:
    none

METHOD  superpane.
description:
    Set the receiver's topPane
abstract:
    no
precondition:
    none
postcondition:
    none
input:
    aPane IS a TopPane

```

```

output:
    none
obligation:
    none

METHOD topPane
description:
    Answer the receiver's topPane.
abstract:
    no
precondition:
    none
postcondition:
    none
input:
    none
output:
    a TopPane
obligation:
    none

METHOD update
description:
    The data being presented may have changed
    so update the contents of the pane.
abstract:
    yes
precondition:
    isOpen
postcondition:
    none
input:
    none
output:
    none
obligation:
    none

METHOD update:
description:
    The data being presented may have changed.
    so update the contents of the pane if the
    specified name matches the pane's.
abstract:
    no
precondition:
    isOpen
postcondition:
    none
input:
    aSymbol IS a Symbol
output:
    none
obligation:
    none

METHOD update:with:
description:
    The data being presented may have changed.
    so update the contents of the pane if the
    specified name matches the pane's.
abstract:
    no
precondition:
    isOpen

```



```

postcondition:
    none
input:
    (nameSymbol IS a Symbol)
    & (aSymbol IS a Symbol)
output:
    none
obligation:
    none

METHOD update:with:with:
description:
    The data being presented may have changed.
    so update the contents of the pane if the
    specified name matches the pane's.
abstract:
    no
precondition:
    isOpen
postcondition:
    none
input:
    (nameSymbol IS a Symbol)
    & (aSymbol IS a Symbol)
    & (anObject IS an Object)
output:
    none
obligation:
    none

METHOD initialize
description:
    Initialize the receiver
abstract:
    no
precondition:
    "initialized"
postcondition:
    initialized
input:
    none
output:
    none
obligation:
    none

```

A.4 NewDispatcher Class

```

description:
    Dispatchers are responsible for processing input for their
    associated panes.
abstract:
    yes
variables:
    pane IS a Pane .
    active IS a Boolean .      "true if the dispatcher is
                                active and processing input"

```

A.5 NewDispatcher Methods

```

METHOD activate

```

```

postcondition:
    none
input:
    (nameSymbol IS a Symbol)
    & (aSymbol IS a Symbol)
output:
    none
obligation:
    none

METHOD update:with:with:
description:
    The data being presented may have changed.
    so update the contents of the pane if the
    specified name matches the pane's.
abstract:
    no
precondition:
    isOpen
postcondition:
    none
input:
    (nameSymbol IS a Symbol)
    & (aSymbol IS a Symbol)
    & (anObject IS an Object)
output:
    none
obligation:
    none

METHOD initialize
description:
    Initialize the receiver.
abstract:
    no
precondition:
    ~initialized
postcondition:
    initialized
input:
    none
output:
    none
obligation:
    none

```

A.4 NewDispatcher Class

```

description:
    Dispatchers are responsible for processing input for their
    associated panes.
abstract:
    yes
variables:
    pane IS a Pane .
    active IS a Boolean .      "true if the dispatcher is
                                active and processing input"

```

A.5 NewDispatcher Methods

```

METHOD activate

```

```

description:
    Make the receiver active to indicate that
    the cursor is in the associated pane and hence the
    pane now has control.
abstract:
    no
precondition:
    isControlWanted
postcondition:
    active
input:
    none
output:
    none
obligation:
    deactivate

METHOD active
description:
    Answer true if the receiver is in an active state
    otherwise answer false.
abstract:
    no
precondition:
    none
postcondition:
    none
input:
    none
output:
    true          "receiver is active"
    | false       "receiver is not active"
obligation:
    none

METHOD cycle
description:
    Cycle to another window
abstract:
    no
precondition:
    none
postcondition:
    ~active
input:
    none
output:
    none
obligation:
    none

METHOD cyclePane
description:
    Cycle to another pane/dispatcher pair in the window.
abstract:
    no
precondition:
    none
postcondition:
    ~active
input:
    none
output:
    none
obligation:

```

none

METHOD deactivate

description:

Make the receiver inactive to indicate that the cursor is no longer in the associated pane and hence the pane does not have control.

abstract:

no

precondition:

active & ~isControlWanted

postcondition:

~active

input:

none

output:

none

obligation:

none

METHOD doesNotHandle

description:

A user action is not serviceable, sound a bell.

abstract:

no

precondition:

none

postcondition:

none

input:

none

output:

none

obligation:

none

METHOD forceCursor

description:

Force the mouse cursor to the top left corner of the associated pane.

abstract:

no

precondition:

active

postcondition:

none

input:

none

output:

none

obligation:

none

METHOD homeCursor

description:

Home the mouse cursor to the top left corner of the associated pane.

abstract:

no

precondition:

active

postcondition:

none

input:

none

```

output:
    none
obligation:
    none

METHOD isControlActive
description:
    Answer true if the cursor is inside the pane,
    and the control is still active otherwise answer false.
abstract:
    no
precondition:
    none
postcondition:
    none
input:
    none
output:
    true          "cursor inside pane and active"
    | false       "cursor not inside pane or not active"
obligation:
    none

```

```

METHOD isControlWanted
description:
    Answer true if the receiver wants control of
    input. Otherwise answer false.
abstract:
    no
precondition:
    none
postcondition:
    none
input:
    none
output:
    true          "control is wanted"
    | false       "control is not wanted"
obligation:
    none

```

```

METHOD modified
description:
    Answer true if the contents of the pane have
    been modified and not saved. Otherwise answer
    false.
abstract:
    yes
precondition:
    none
postcondition:
    none
input:
    none
output:
    true          "panes has unsaved changes"
    | false       "no unsaved changes to pane"
obligation:
    none

```

```

METHOD pane
description:
    Answer the receiver's pane
abstract:
    no

```

precondition:

none

postcondition:

none

input:

none

output:

a Pane

obligation:

none

METHOD pane:

description:

Set the receiver's pane to be the given pane.

abstract:

no

precondition:

~open

postcondition:

none

input:

aPane IS a Pane

output:

none

obligation:

none

METHOD select

description:

The user issued a select operation (eg. left mouse).

abstract:

yes

precondition:

active

postcondition:

none

input:

none

output:

none

obligation:

none

METHOD topDispatcher

description:

Answer the dispatcher of the top pane

abstract:

no

precondition:

none

postcondition:

none

input:

none

output:

a TopPane

obligation:

none

METHOD initialize

description:

Initialize the receiver.

abstract:

no

precondition:

```

    ^initialized
postcondition:
    initialized
input:
    none
output:
    none
obligation:
    none

METHOD processControlKey:
description:
    Process the aCharacter as a control key.
abstract:
    yes
precondition:
    active
postcondition:
    none
input:
    (aCharacter IS a Character)
    & (aCharacter isControlKey)
output:
    none
obligation:
    none

METHOD processFunctionKey:
description:
    Process the aCharacter as a function key.
abstract:
    no
precondition:
    active
postcondition:
    none
input:
    (aCharacter IS a Character)
    & (aCharacter isFunctionKey)
output:
    none
obligation:
    none

METHOD processInput
description:
    Keep processing characters until the cursor is
    no longer in the pane.
abstract:
    no
precondition:
    ^active
postcondition:
    ^active
input:
    none
output:
    none
obligation:
    none

METHOD processInputKey:
description:
    Process the aCharacter as regular input key.
abstract:

```

```
yes
precondition:
  active
postcondition:
  none
input:
  aCharacter IS a Character
output:
  none
obligation:
  none

METHOD processKey:
description:
  Analyze the given character and process it accordingly.
abstract:
  no
precondition:
  none
postcondition:
  none
input:
  aCharacter IS a Character
output:
  true      "receiver is active"
  | false   "receiver is not active"
obligation:
  none

METHOD processLastInput:
description:
  Process the last character that was waiting when
  the receiver activated.
abstract:
  no
precondition:
  none
postcondition:
  none
input:
  aCharacter IS a Character
output:
  none
obligation:
  none
```


B Appendix B: Subpane/Dispatcher Group

In this section we present the SubPane/Dispatcher group. Definitions for the group and its elements are provided in the following order:

- The SubPane/Dispatcher group.
- The NewSubPane class and its methods.

B.1 SubPaneDispatcher Group

```

name:
  SubPaneDispatcher
descriptiveName:
  SubPane/Dispatcher Framework
description:
  with super and;

  The area occupied by a subpane is controlled by its
  framingBlock which can either be specified as a rectangle
  or in terms of ratios describing the proportions of
  the subpane within the window described by the topPane.
abstract:
  yes
inheritsFrom:
  PaneDispatcher
defines participants:
  with super and;
  pane IS a SubPane .
uses:
secondary participants:

synonyms

invariant:

behaviour:
paths:
  with super and;

  create
    model >>
      super create &
        (framingBlock: | framingRatio:).
exception paths:

```

B.2 NewSubPane Class

```

description:
  with super and;
  Subpanes occupy an area described by their framing block.
abstract:
  yes
variables:
  with super and;
  framingBlock IS a Context.
  changeSelector IS a Symbol.
  topCorner IS an Integer.
  scrollbar IS a BitBlt.

```

B.3 NewSubPane Methods

METHOD change

description:

Answer the selector to send to the model in the event that it should be notified of an important changed in the receiver. Answer nil if the model does not want to be notified of changes.

abstract:

no

precondition:

none

postcondition:

none

input:

none

output:

a Symbol

| nil "model does not want to be notified of important changes"

obligation:

none

METHOD change:

description:

Set the selector to send to the model in the event that it should be notified of an important changed in the receiver. Nil indicates the model does not want to be notified of changes.

abstract:

no

precondition:

none

postcondition:

none

input:

(aSymbol IS a Symbol) | (aSymbol isNil)

output:

none

obligation:

none

METHOD displayWindow

description:

abstract:

no

precondition:

postcondition:

input:

output:

obligation:

METHOD framingBlock:

description:

Set the framing block which describes the frame of the receiver.

abstract:

no

precondition:

none

postcondition:

none

input:

aBlock IS a Context

output:

none

obligation:
none

METHOD framingRatio:

description:
Set the rectangle which describes the
frame of the receiver in terms of ratios to
the window (topPane) it is contained in.

abstract:

no

precondition:

none

postcondition:

none

input:

aRectangle IS a Rectangle

output:

none

obligation:

none

METHOD reframe:

description:

abstract:

no

precondition:

postcondition:

frame REFLECTS aRectangle
& paneScanner frame REFLECTS frame

input:

output:

obligation:

METHOD showWindow

description:

Show the current contents of the receiver,
including a border and the current state of
the selection cues.

It is possible the window is not active
hence the selection may be grayed.

abstract:

no

precondition:

isOpen

postcondition:

none

input:

none

output:

a Symbol

obligation:

none

METHOD initialize

description:

Initialize the receiver.

abstract:

no

precondition:

"initialized

postcondition:

initialized

input:

none

output:

```
none
obligation:
none

METHOD topCorner
description:
    Answer the relative offset of the top left corner
    of the data currently displayed by the receiver.
abstract:
    no
precondition:
    none
postcondition:
    none
input:
    none
output:
    a Point
obligation:
    none
```

C Appendix C: TopPane/TopDispatcher Group

In this section we present the TopPane/TopDispatcher group. Definitions for the group and its elements are provided in the following order:

- The TopPane/TopDispatcher group.
- The NewTopPane class and its methods.
- The NewTopDispatcher class and its methods.

C.1 TopPaneDispatcher Group

```

name
  TopPaneDispatcher
descriptiveName:
  TopPane/Dispatcher Framework
abstract:
  no
description:
  with super and;

  TopPane's control the areas of the window not occupied
  by any pane, this includes the label and its icons.

  TopPanes are responsible for ensuring that the subpanes
  are all opened, closed, reframed and notified of
  window activations and deactivations

inheritsFrom:
  PaneDispatcher
defines participants:
  with super and;
  pane IS a TopPane.
  dispatcher IS a TopDispatcher .
uses:

secondary participants:
  with super and,
  subpanes IS LIKE COLLECTION OF a SubPane.

synonyms
  topPane = pane
  topDispatcher = dispatcher.
  <.V. subpanes> toppane = pane .

  subpaneDispatchers = <.V. subpanes> dispatcher .

invariant:
  with super and,
  (< V. subpanes> frame REFLECTS topPane frame) USING topPane reframe:.

behaviour.
  create ;
  schedule ;
  (windowActivating;
    (update*;
```

```

        cursorEntersPane;
        (update | userInput | collapse)+;
        cursorLeavesPane;
    );
    windowDeactivating;
)*
terminate.
paths:
    super ;

create
    model >>
        super create &
        ([pane label:] &
        [leftIcons:] &
        [rightIcons:] &
        pane addSubpane:*)
    .

open:
    model >>
        pane reframe;;
        <.V. subpanes> reframe;;
        pane open;
        <.V. subpanes> open .

close:
    (USER | model) >>
        (dispatcher closeWindow | dispatcherCloseIt) ;
        <.V. subpanes> close;
        [model closingWindow];
        pane release.

schedule
    open;
    scheduleWindow .

refresh:
    (pane | model) >>
        super refresh;
        model label;
        (Scheduler resume |
        (pane displayLabel; pane highlightLabel)) .

windowActivating:
    USER >>
        pane displayWindow;
        [model showWindow];
        pane displayLabel;
        pane highlightLabel;
        pane displayLabelIcons;
        pane showWindow;
        <.V. subpanes> displayWindow;
        <.V. subpanes> showWindow;
        [model reactivateWindow].

windowDeactivating:
    USER >>
        pane graySelection;
        <.V. subpanes> graySelection .

cursorEntersPane:
    USER >>
        (super cursorEntersPane
        )

```

```

| <.E. subpaneDispatchers> processInput .

cursorLeavesPane:
  USER >>
  <.E. subpaneDispatchers> processInput |
  (dispatcher cursorOut:)* .
exception paths:

```

C.2 NewTopPane Class

```

description:
  with super and;
  A TopPane id responsible for coordinating the frames
  of all subpanes.
abstract:
  no
variables:
  with super and;

  dispatcher IS a TopDispatcher .
  subpanes IS COLLECTION OF a SubPane .
  label IS a String .
  collapsed IS a Boolean.
  minimumSize IS an Integer .
  borders IS a Form.
  foreColor IS a Form .
  backColor IS a Form .
  previousFrame IS a Rectangle.
  iconArray IS an Array .

```

C.3 NewTopPane Methods

```

METHOD activateWindow
description:
  with super and;
  The pane that should be active is forced
  to begin processing.
precondition:
  isOpen
  & ~isWindowActive
postcondition:
  isWindowActive
input:
output:
obligation:

METHOD addSubpane:
description:
  Add the given SubPane to the receiver's collection
  of subpanes.
abstract:
  no
precondition:
  ~isOpen
postcondition:
  none
input:
  aPane IS a SubPane
output:
  none
obligation:
  none

```

METHOD close
description:
 super ;
 Inform all of the subpanes that the window is closing.
abstract:
precondition:
postcondition:
input:
output:
obligation:

METHOD cycle
description:
 Cycle to another window
abstract:
 no
precondition:
 isWindowActive
postcondition:
 ¬isWindowActive
input:
 none
output:
 none
obligation:
 none

METHOD deactivatePane
description:
 Do nothing.
abstract:
precondition:
postcondition:
input:
output:
obligation:

METHOD deactivateWindow
description:
 The window (top pane) is activating, change any
 visual clues for the window.
precondition:
 isOpen
 & ¬isWindowActive
postcondition:
 ¬isWindowActive
input:
 none
output:
 none
obligation:
 none

METHOD defaultDispatcherClass
description:
abstract:
 no
precondition:
postcondition:
input:
output:
 TopDispatcher
obligation:

METHOD displayWindow**description:**

with super and;

Inform all subpanes to display themselves.

abstract:**precondition:****postcondition:****input:****output:****obligation:****METHOD graySelection****description:**

super ;

Inform each of the subpanes that a deactivation is occurring and they should gray their visual cues.

abstract:

no

precondition:

super

& isWindowActive

postcondition:**input:****output:****obligation:****METHOD hasCursor****description:****abstract:****precondition:****postcondition:****input:****output:****obligation:****METHOD highlightLabel****description:**

Highlight the label to indicate the receiver window is active.

abstract:

no

precondition:

!isWindowActive

postcondition:

none

input:

none

output:

none

obligation:

none

METHOD label**description:**

Answer the receiver's label

abstract:

no

precondition:

none

postcondition:

none

input:

none

output:

a String
 obligation:
 none

METHOD label:
 description:
 Set the receiver's label to aString.
 abstract:
 no
 precondition:
 none
 postcondition:
 none
 input:
 aString IS a String
 output:
 none
 obligation:
 none

METHOD leftIcons:
 description:
 Set the array containing the names of icons
 to display at the left side to be anArray
 abstract:
 no
 precondition:
 none
 postcondition:
 none
 input:
 anArray IS an Array
 output:
 none
 obligation:
 none

METHOD minimumSize
 description:
 Set the minimum size allowed for the window.
 abstract:
 no
 precondition:
 none
 postcondition:
 none
 input:
 none
 output:
 a Point
 obligation:
 none

METHOD minimumSize:
 description:
 Set the minimum size allowed for the window.
 abstract:
 no
 precondition:
 none
 postcondition:
 none
 input:
 aPoint IS a Point
 output:

```

    none
obligation:
    none

METHOD open
description:
    super ;
    Instruct all of the subpanes to open.
abstract:
    no
precondition:
    ~isOpen
    & ~active
    & name notNil
    & model notNil
postcondition:
    isOpen &
    <.V. subpanes> isOpen
input:
output:
obligation:

METHOD performMenu
description:
abstract:
precondition:
postcondition:
input:
output:
obligation:

METHOD reframe:
description:
    with super and;
    Reframe all of the subpanes.
abstract:
    no
precondition:
postcondition:
    <.V. subpanes> frame REFLECTS frame
input:
output:
obligation:

METHOD rightIcons:
description:
    Set the array containing the names of icons
    to display at the right side to be anArray
abstract:
    no
precondition
    none
postcondition:
    none
input:
    anArray IS an Array
output:
    none
obligation:
    none

METHOD select
description:
    The user issued a select operation (eg. left mouse).
    If an icon has been selected answer its name

```

```

    otherwise nil.
precondition:
    isWindowActive &
    active
postcondition:
    none
input:
    none
output:
    a Symbol |
    nil
obligation:
    none

```

```

METHOD subpanes
description:
    Answer the collection of the receiver's SubPanels
abstract:
    no
precondition:
    none
postcondition:
    none
input:
    none
output:
    an OrderedCollection
obligation:
    none

```

```

METHOD topPane
description:
abstract:
precondition:
postcondition:
input:
output:
obligation:

```

```

METHOD collapse
description:
    Collapse the window into an icon.
abstract:
    no
precondition:
    isOpen
    & isWindowActive
    & ~collapsed
postcondition:
    collapsed
input:
    none
output:
    none
obligation:
    uncollapse

```

```

METHOD collapsed
description:
    Answer true if the receiver is collapsed
    otherwise answer false.
abstract:
    no
precondition:
    none

```

```

postcondition:
    none
input:
    none
output:
    true      "receiver is collapsed"
    | false   "receiver is not collapsed"
obligation:
    none

```

```

METHOD createIcons:
description:
    Create icons with the names given in anArray.
abstract:
    no
precondition:
    none
postcondition:
    none
input:
    anArray IS an Array
output:
    an Array
obligation:
    none

```

```

METHOD cyclePane:
description:
    Move to the pane following aPane.
abstract:
    no
precondition:
    isOpen
    & isWindowActive
postcondition:
    none
input:
    none
output:
    none
obligation:
    none

```

```

METHOD displayLabel
description:
    Display the receiver's window label
precondition:
    isOpen
    & isWindowActive
postcondition:
    none
input:
    none
output:
    none
obligation:
    none

```

```

METHOD displayLabelIcons
description:
    Display the receiver's window label icons
abstract:
    no
precondition:
    isOpen

```

```
    & isWindowActive
postcondition:
    none
input:
    none
output:
    none
obligation:
    none
```

```
METHOD initialize
description:
abstract:
precondition:
postcondition:
input:
output:
obligation:
```

```
METHOD labelFrame
description:
    Answer the receiver's frame for the label.
precondition:
    none
postcondition:
    none
input:
    none
output:
    a Rectangle
obligation:
    none
```

```
METHOD move:
description:
    The user has asked to move the window.
abstract:
    no
precondition:
    isOpen
    & isWindowActive
postcondition:
    none
input:
    none
output:
    none
obligation:
    none
```

```
METHOD reframe
description:
    Resize the window from user feedback.
abstract:
    no
precondition:
    isOpen
    & isWindowActive
postcondition:
    none
input:
    none
output:
    none
obligation:
```

none

METHOD reframeLabel

description:

Resize the label using the current frame.

abstract:

no

precondition:

isOpen

& isWindowActive

postcondition:

none

input:

none

output:

none

obligation:

none

METHOD resize:

description:

Resize the window to occupy the given rectangle with its frame.

abstract:

no

precondition:

isOpen

& isWindowActive

postcondition:

none

input:

aRectangle IS a Rectangle

output:

none

obligation:

none

METHOD searchForActivePane

description:

Answer the subpane that wants control.

abstract:

no

precondition:

none

postcondition:

none

input:

none

output:

a SubPane

| nil

obligation:

none

METHOD textModified

description:

Answer true if the receiver contains any unsaved modifications. Each of the subpanes is checked.

precondition:

none

postcondition:

none

input:

none

output:

```

    true          "receiver has unsaved changes"
    | false       "receiver has not unsaved changes"
obligation:
    none

```

METHOD uncollapse

```

description:
    Uncollapse the window into an icon.

```

```

abstract:
    no
precondition:
    isOpen
    & isWindowActive
    & collapsed
postcondition:
    ~collapsed
input:
    none
output:
    none
obligation:
    none

```

METHOD windowFrame

```

description:
    Answer the receiver's frame, answer
    nil if the receiver does not have a frame yet.

```

```

abstract:
    no
precondition:
    none
postcondition:
    none
input:
    none
output:
    a Rectangle
obligation:
    none

```

C.4 NewTopDispatcher Class

```

description:
    with super and;
    A TopDispatcher processes input for the outer
    label and whenever the cursor is not in a subpane.
abstract:
    no
variables:
    with super and;

    pane IS a TopPane .

```

C.5 NewTopDispatcher Methods

METHOD activateWindow

```

description:
    The window (top pane) is activating, change any
    visual clues for the window. Then, find
    the pane that should be active and force

```



```

    it to begin processing.
abstract:
    no
precondition:
    isOpen & ~isWindowActive
postcondition:
    isWindowActive
input:
    none
output:
    none
obligation:
    deactivateWindow

METHOD   closeIt
description:
    Close the window and inform the schedule that
    it should schedule another window. Before
    closing the window check for any unsaved changes.
abstract:
    no
precondition:
    isOpen
postcondition:
    ~isOpen & ~isWindowActive
input:
    none
output:
    none
obligation:
    none

METHOD   closeWindow
description:
    Close the window and inform the schedule that
    it should schedule another window. Before
    closing the window check for any unsaved changes.
abstract:
    no
precondition:
    isOpen
postcondition:
    ~isOpen & ~isWindowActive
input:
    none
output:
    none
obligation:
    none

METHOD   deactivateWindow
description:
    The window (top pane) is activating, change any
    visual clues for the window. Then, find
    the pane that should be active and force
    it to begin processing.
abstract:
    no
precondition:
    isOpen & ~isWindowActive
postcondition:
    ~isWindowActive
input:
    none
output:

```

none
 obligation:
 none

METHOD display

description:
 Display the current contents of the window.
 abstract:
 no
 precondition:
 isOpen
 postcondition:
 none
 input:
 none
 output:
 none
 obligation:
 none

METHOD isControlActive

description:
 with super and;
 Control is wanted if no subpane wants control.
 abstract:
 no
 precondition:
 postcondition:
 input:
 output:
 obligation:

METHOD label

description:
 Answer the label of the window
 abstract:
 no
 precondition:
 none
 postcondition:
 none
 input:
 none
 output:
 a String
 obligation:
 none

METHOD open

description:
 Open the window (but do not schedule it)
 Open all of the subpanes
 abstract:
 no
 precondition:
 ~isOpen & ~isWindowActive
 postcondition:
 isOpen & ~isWindowActive
 input:
 none
 output:
 none
 obligation:
 none

```

METHOD  openIn:
description:
    Open the window (but do not schedule it) using
    the given rectangle for the frame.
    Open all of the subpanes
abstract:
    no
precondition:
    ~isOpen & ~isWindowActive
postcondition:
    isOpen & ~isWindowActive
input:
    aRectangle IS a Rectangle
output:
    none
obligation:
    none

METHOD  openWindow
description:
    Close the receiver and release any
abstract:
    no
precondition:
    isOpen
postcondition:
    none
input:
    none
output:
    none
obligation:
    none

METHOD  scheduleWindow
description:
    Schedule the window by forwarding the request to
    the scheduler
abstract:
    no
precondition:
    isOpen & ~isWindowActive
postcondition:
    isWindowActive
input:
    none
output:
    none
obligation:
    none

METHOD  searchForActivePane
description:
    Search for the subpane which needs control and
    tell its dispatcher to begin processing.
abstract:
    no
precondition:
    isOpen & isWindowActive
postcondition:
    none
input:
    none
output:
    none

```

obligation:
none

METHOD select

description:
The user issued a select operation (eg. left mouse)
Check if its in any icons and service them.
Also check to see if the user is attempting to
move the window by dragging the receiver's label.

abstract:
no

precondition:
postcondition:
input:
output:
obligation:

METHOD close

description:
Close the receiver and release any

abstract:
no

precondition:
isOpen
postcondition:
none

input:
none
output:
none
obligation:
none

METHOD collapse

description:
Collapse the window into an icon. If the receiver
is already collapsed then uncollapse it

abstract:
no

precondition:
isOpen & isWindowActive
postcondition:
none

input:
none
output:
none
obligation:
none

METHOD cursorOut:

description:
Process the input character while the cursor is outside
the window.

abstract:
no

precondition:
isOpen
& isWindowActive
& isActive

postcondition:
none

input:
aCharacter IS a Character
output:

none
obligation:
none

METHOD defaultMenu
description:
 Answer the default window menu.
abstract:
 no
precondition:
 none
postcondition:
 none
input:
 none
output:
 a HierarchicalMenu
obligation:
 none

METHOD defaultOpenSize
description:
 Answer the default size to open the receiver.
 Obtain the default from the model if possible.
abstract:
 no
precondition:
 none
postcondition:
 none
input:
 none
output:
 a Point
obligation:
 none

METHOD initWindowSize
description:
 Answer the possible initial size to open the receiver.
abstract:
 no
precondition:
 none
postcondition:
 none
input:
 none
output:
 a Point
obligation:
 none

METHOD move
description:
 The user has asked to move the window.
abstract:
 no
precondition:
 isOpen
 & isActive
postcondition:
 none
input:
 none

```
output:
    none
obligation:
    none

METHOD  openFrame
description:
    Answer the frame to use for when opening the receiver.
abstract:
    no
precondition:
    ~isOpen
postcondition:
    none
input:
    none
output:
    none
obligation:
    none

METHOD  resize
description:
    Resize the window from user feedback.
abstract:
    no
precondition:
    isOpen
    & isWindowActive
postcondition:
    none
input:
    none
output:
    none
obligation:
    none
```

D Appendix D: Sample Window Group

In this section we present the Sample Window group. Definitions for the group and its elements are provided in the following order:

- The SampleWindow group.
- The SampleModel class and its methods.

D.1 SampleWindowFramework Group

```

name:
    SampleWindow
descriptiveName:
    Sample Window
description:
    A window with a toppane and two subpanes.
abstract:
    no
inheritsFrom:
    none
defines participants:
    model IS a SampleModel.
uses:
    window = 1 OF TopPaneDispatcher .
    fun = 1 OF TextPaneDispatcher.
    boring = 1 OF TextPaneDispatcher.
secondary participants:
    none
synonyms:
    fun topPane = window pane .
    boring topPane = window pane.

    fun model = model
    boring model = model.
    window model = model.
invariants:
    (fun REFLECTS model funText) USING fun update.
    & (boring REFLECTS model boringText) USING boring update.
behaviour:
    windowCreate;
    window schedule;
    ((window windowActivating;
      ((fun saving;boring update) | boring saving)*
      window deactivatingWindow
    )*)
    [model closeSampleWindow];
    window closeIt .
paths:
    windowCreate
        model>>
            fun create;
            boring create;
            window create .
exception paths:
    none

```

D.2 SampleModel Class

```
description:
    A demonstration window which uses to sub groups.
abstract:
    no
variables:
    toppane is a TopPane.
```

D.3 SampleModel Methods

```
METHOD acceptBoringText:from:
description:
    Accept changes to the text and answer true
    to indicate they were accepted.
abstract:
    no
precondition:
    isOpen
postcondition:
    none
input:
    none
output:
    true
obligation:
    none
```

```
METHOD acceptFunText:from:
description:
    Accept changes to the text and answer true
    to indicate they were accepted
abstract:
    no
precondition:
    isOpen
postcondition:
    none
input:
    none
output:
    true
obligation:
    none
```

```
METHOD boringText
description:
    Answer the boring text.
abstract:
    no
precondition:
    none
postcondition:
    none
input:
    none
output:
    a String
obligation:
    none
```

```
METHOD closeSampleWindow
description:
    Create and open the window.
abstract:
```



```

    no
precondition:
    isOpen
postcondition:
    ~isOpen
input:
    none
output:
    none
obligation:
    none

METHOD funText
description:
    Answer the fun text.
abstract:
    no
precondition:
    none
postcondition:
    none
input:
    none
output:
    a String
obligation:
    none

METHOD isOpen
description:
    Answer true if the window is open
    otherwise answer false.
abstract:
    no
precondition:
    none
postcondition:
    none
input:
    none
output:
    true      "window is open"
    | false   "window is closed"
obligation:
    none

METHOD openSampleWindow
description:
    Create and open the window
abstract:
    no
precondition:
    ~isOpen
postcondition:
    isOpen
input:
    none
output:
    none
obligation:
    none

```

END

2/3·12·92

FIN