A Role-Based Architectural Model Applied to Object-Oriented Systems

by

Ronald S. Casselman, B.Eng.

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

Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6
August 20, 1993

Abstract

This thesis describes the *role-based model* for software architecture applied to object-oriented systems and frameworks. The purpose of the role-based model is to separate the architecture of a software system into parts that may be considered independently or composed and viewed as elements of a larger whole. The model factors a system into groups of interacting objects, called *role teams*. A *role team* consists of nested role teams and *roles*, which describe the properties required of objects to participate in the role team. An object is represented in the system by the *role* or *roles* it plays in the context of a role model. The behaviour of a role team is expressed as a causal sequence of activities; role players (objects) are responsible for performing the activities. The structure of a system is the composition of possibly many role teams, and the overall behaviour of a system results from linking the behaviour of the individual teams in a causal sequence. Objects play roles in multiple teams subject to structural and temporal constraints. Structural constraints limit which roles an object may play and temporal constraints limit when an object may play a role. The role-based model is particularly useful for representing systems that have dynamic structure, e.g., object creation, destruction, and movement. Models of dynamic structure appear to missing in current object-oriented techniques.

Informal visual notations are used to specify systems in the role-based model. An architectural notation is developed as part of this thesis. System behaviour is described using a subset of Buhr's timethreads notation. The goal is to capture the essential system architecture and large recurring behaviour patterns as these form the basis for understanding more detailed issues.

The role-based model is developed by extending the concepts of a more general metamodel. The role-based model is applied to an existing object-oriented framework for creating semantic graphic editors (HotDraw).

Acknowledgements

This thesis has taken too long to finish. Almost everyone on this list has told me to, “just get it done”. I ignored them every time.

First and foremost I must thank Ray Buhr who has been my supervisor, colleague, boss, mentor, supplier of funds, and friend for the past three years. I have learnt more from Ray than I can measure. He has expanded my horizons in both my professional and my personal lives, and I am better for it. I can only hope that someday I can reach a point where I can do the same for someone else.

Gerald Karam, my co-supervisor, acted as a source of objective-rational thought throughout this project. In the scope of this thesis, Gerald forced me to be precise when I needed to be and the result was a much better product. Outside the scope of this thesis, Gerald has been a long-time friend and advisor; I will always be indebted to Gerald for giving me the opportunity to “come home”.

Members of the Architecture Handbook Group headed by Bruce Anderson have inspired me and given me encouragement. I list only some of the names but thank all the “handbookers”. Richard Helm listened when the concepts were muddy, suggested directions, and gave me confidence that the ideas were interesting. Doug Lea gave me the opportunity to see object-oriented design first hand and offered comments on a draft of the thesis. Desmond Desouza thought the ideas were interesting and suggested ways that they could be further formalized. Ralph Johnson suggested that I model HotDraw.

Bran Selic and everyone at ObjecTime deserve mention because so many of their ideas have influenced this work.

Mike Petras of Bell Northern Research (BNR) has been a willing source of technical expertise as the industrial contact on the DOORS (Design of Object-Oriented Real-time Systems) project.

The students, staff, and others in the “Carleton crowd” is too long to list, but a few individuals stand out. Greg Franks probably told me more than anyone else to “just get it done”, he also gave me excellent editorial comments on my scattered writing style. Francois Pomerleau listened for hours to my babbling about roles and meta-models and timethreads, etc.; he has been a close compatriot and friend. John Bryant dug deep into the case study and found several logical inconsistencies. Many students in Ray’s graduate course suffered through an early draft of this material.

I would like to thank my family. Mom and Dad are always behind me to pick up the slack anyway they can. Diane, you have shared every anguish and triumph; you deserve this thesis as much as I do. Amanda and Rebecca, I hope I say, “just a minute honey” much less in the future.

I gratefully acknowledge the financial support I have received from Bell Northern Research (BNR) and the Natural Sciences and Engineering Research Council (NSERC).

Contents

List of Figures

Glossary

Activity: The carrying out of a responsibility, the details of which are suppressed in an architectural model.

Architectural Entity: A uniquely identifiable element of a system's architecture.

Architectural Meta-Model: A generalized description of the properties of software architectural models with emphasis given to dynamic structure.

Architecture: Of a software system is a conceptual model of its operation and organization in terms of its components and the communication pathways among the components. There are two related aspects to software architecture: structure and behaviour.

Behaviour: The temporal sequencing of activities performed by a system and the communication patterns among its components that are necessary for the system to achieve its objectives.

Causality-flow: A causally connected sequence of activities that occur through the architecture of a system.

Collaboration Group: A group of objects participating to achieve some objective. An instance of a role team.

Component: The sources of activity and/or the units of organization that form a designer's conceptual model of a system's architecture. (optionally refered to as operational component)

Composite Lifetime: The time for which a component is operational and exists within a system.

Component-Bound Role: A role which is bound to one component (object) for the operational lifetime of the role.

Communication pathway: A logical pathway among components over which messages flow. A communication pathway may be achieved by a single wire or may be decomposed into a wiring of components.

Container: A component with nested subcomponents.

Data flow: A value that flows over a wire (communication pathway) or along a timethread. Data flows are contained in messages.

Data Value: A data abstraction. A data value is distinguished from an object because data values do not play roles in a role-based architectural model.

Dynamic Structure: The changes that occur to a system's structure while it is running; e.g., object creation, destruction, and movement.

Equivalence: A relationship between places that specifies that the same components will occupy the joined places at runtime.

Fixed Component: A component that operates in a container for the composite lifetime of the container.

Framework: A collection of related classes that serve as a skeletal template for creating new applications in a given problem domain (constructive view). Frameworks embody a role-based architecture implemented in a distributed fashion by the code of the classes (architectural view).

Instance flow: The flow of an object's identifier over a wire or along a timethread. Conceptually, an instance may refer to the *movement* of a object when the flow supports the installation of an object in a role. Instance flows are contained in messages.

Message: A container of information that flows over a wire.

Multiple Part-of: The ability of a component to be involved in multiple decompositions (i.e., places), perhaps simultaneously.

Object: A component of a role-based model. Objects appear as the attributes of roles and they play roles in role teams.

Object Architecture: A basic architectural view in terms of objects and object references.

Object Reference: A direct connection between two objects that is achieved through the exchange of object identifiers.

Operational Lifetime: The period of time that a component is operational in a single container.

Optional Component: A component that begins to operate in a container sometime after the container begins to operate. An optional component may end its operation in a container before the container ends its own operation.

Place: A node in an architectural wiring diagram. A conceptual location in a software architecture at which activities are performed.

Placeholder: A reserved place in the architecture of a system that may be filled dynamically with different components through time.

Placeholder Role: A role which may be filled dynamically with different objects through time.

Primitive Component: A component which may not be decomposed into subcomponents at the particular level of abstraction being modelled.

Role: A role is a place in a role architecture. A role is an element of a role-based model; it describes the properties of an object necessary for it to participate in the role team that contains the role. Objects play roles in role teams.

Role Architecture: An architectural model for software systems that is capable of expressing dynamic structure.

Role Team: A component in a role architecture that may contain roles and nested role

teams.

Role Method: A primitive component of a role that performs an activity when sent a message.

Role Wire: A wire between roles over which messages flow.

Structure: The decomposition of a system into its components, the composition relationships among the components, and the communication pathways among the components to achieve interaction. Structure alone does not include system operation (i.e., behaviour) and is a different concept than architecture.

Timethread: A visual representation of a causal-flow path. Visually a timethread is a path traced over a view of a system to show the relationship between the activities performed and the system's architecture.

Terminal: A connection point for wires. Terminals may exist on the interface of components or appear as routers at points where wires meet.

Wire: An undirectional conduit over which messages flow. A concrete connection among components that may not be further decomposed into sub-wirings.

Chapter 1: Introduction

1.1 Software Architecture and Collaboration Groups

The architecture of a software system is a conceptual model of its operation and organization in terms of its components and their communication pathways. Garlan [21] defines the term architecture as:

> *"a collection of computational components–or simply components–together with a description of the interactions between these components–the connectors."*

Buhr [13] offers a similar definition but adds the notion of system operation when he defines architecture as:

> *"the nature of the operational components..., their operational interactions (e.g., exchanging data, transferring control), and how they operate together as a whole to accomplish the overall purposes of the system."*

Components (meaning *operational* components (Buhr) or *computational* components (Garlan)) are the sources of activity and/or units of organization of a system's architecture; they are elements that form a designer's mental model of a system in operation. Examples of components include subsystems, layers, objects, and processes. Components may come from the underlying support environment (e.g., programming language or runtime system) or not, in which case the components are conceptual but still important for understanding system operation at the architectural level. *Communication pathways* (meaning *operation-*

al interactions (Buhr) or *connectors* (Garlan)) can be varied and include procedure calls, messaging, and event broadcasts. Like components, some communication pathways may be conceptual meaning that they must be constructed from other elements when they are not explicit in the implementation nor provided by the underlying support environment. For example, a direct communication pathway between two components may be useful for design even though the communication in the implementation is routed through other components.

There are two related aspects to system architecture: *structure* and *behaviour. Structure* is the decomposition of a system into its components, the composition relationships among the components, and the communication pathways among the components to achieve interaction. Graphically, this leads to an architectural description of a system in which the nodes represent the components and the arcs joining the nodes represent the communication pathways. The properties of the nodes and arcs differentiate software models, e.g., data flow diagrams [51], various module and subsystem connection languages and notations [48][53], and object modelling notations [44][17]. Many software structure models represent a system as a static configuration of nodes and arcs. A static structure model may be appropriate for periods of system operation when the structure is fixed or for representing the organization of large-grained components, e.g., subsystems which tend to be configured together in a static way [13]. In general, however, the structure of a software system may be dynamic: components and communication pathways may be created and destroyed, and components may "move" through a system as they cooperate with different components. A dynamic structure model represents the dynamic aspects of a system's structure (components and communication pathways) through time.

Behaviour is the temporal sequencing of the activities performed by a system and the communication patterns among its components that are necessary for the system to achieve its objectives. An activity is the carrying out of a responsibility, the details of which are suppressed in an architectural model. Ultimately the aggregate behaviour of a system is achieved by the activities performed by individual components as they cooperate through interactions over their communication pathways. There are many component-based behaviour description techniques that take this view, e.g., communicating state machines

[45][44], coupled Petri Nets [5], etc. However, designers often use paths through many components as a means of reasoning about system behaviour. Buhr [14] uses the term causality-flow to refer to a sequence of causally connected activities that occurs through a system and across possibly many components. Designers use causality-flow, or something very similar, to drive design during development [31][7][14][49]; and during other development stages, e.g., during re-engineering [31] as a tool for understanding a system's architecture. Causality-flow provides a global representation of behaviour at the same level as structural design notations.

There are many *architectural styles* [21], i.e., standard ways of organizing the elements of a system's architecture. The object-oriented architectural style may be described as a collection of components, called objects, and their interactions. An object is a black-box like component that provides interface methods through which other objects may affect the local state information that it maintains. Objects communicate through the exchange of messages. The result is a cooperative effort among the objects to accomplish a set of objectives. The desire for reuse in object-oriented systems leads to an architectural style that is characterized by dynamically-bound connections between objects and fine-grained interaction patterns (i.e., message exchanges). Dynamically-bound connections mean that objects establish communication pathways dynamically through the exchange of object identifiers. This defers commitment to the identity of communication partners and supports reuse of plug-compatible objects. Fine-grained interaction patterns make object-oriented systems more flexible but also create an organization that is difficult to reason about at the source code level. This makes causality-flow an important concept because representations for causality-flow can abstract from the details of individual object interactions.

Beck and Cummingham [7] write that, "one of the key distinguishing features of object design is that no object is an island. All objects stand in relation to others, on whom they rely for services and control." A group of interdependent objects interacting to achieve some objective is called a *collaboration group* [13][26]. A collaboration group is a component of a system's architecture and some possible examples are: two peer processes in a distributed file transfer system cooperating through the exchange of messages to deliver files in a reliable manner, and two or more figure objects coordinating with one another to ensure

that they remain connected as they are dragged and resized. Several authors have written about the importance of collaboration groups in object-oriented development. Helm [26] notes that throughout an object-oriented system, groups of related objects will often cooperate to perform some task, and that the interaction patterns within a collaboration group are often repeated throughout a system, and even in different systems, with different participating objects. Wirfs-Brock and Johnson [52] observe that reuse in object-oriented systems typically occurs by reusing groups of objects interacting to accomplish tasks.

Collaboration groups represent a reusable architectural abstraction above the level of individual objects that are essential for understanding, designing, and reusing object-oriented systems. The structure of a collaboration group includes its object participants, their structure and interconnections, and any nested collaboration groups. The behaviour of a collaboration group is the sequence of activities performed by the participants (i.e., the causal-flow path through the group) and the temporal order of the messages exchanged among them. For the system as a whole, one needs to understand how the object participants play their roles among possibly many groups, how the groups are sequenced, and how the groups interact to achieve the major causal-flows through the system.

Classes and inheritance, key parts of object-oriented development, are more like construction materials then operational components. They are construction materials because they provide the templates from which objects (the components) are created. Classes and inheritance are not operational concepts in most object-oriented design[1], but must be organized such that the architectural design can be achieved. A *framework* is an organization of classes and inheritance that embodies a system's architecture. From a constructive standpoint, a framework is a set of related classes that may be extended to create new applications in a given problem domain. From an architectural perspective, a framework is a reusable architectural description of a system that embodies collaboration groups and their interaction protocols. Examples of frameworks include: Smalltalk-80's Model/View/Controller framework [22], and the Choices framework for building operating systems [37].

Unfortunately, collaboration groups and the inter-object dependencies they imply are

[1.] Meta-object protocols [40] are the exception.

not explicit in the source code of object-oriented systems and frameworks, but are distributed among, and buried in, the methods of many classes. This makes the existence and understanding of collaboration groups, and the patterns of behaviour they imply, difficult to infer from code. There have been techniques developed for specifying collaboration groups, e.g., Helm's contracts [26] and Booch's mechanisms [8]. Helm's contract notation is formal and textual. A contract is a specification of how the elements of a collaboration group participate to accomplish some objective. Although powerful, formal textually-based notations are often difficult to use as tools for reasoning and for communicating the essence of a system's architecture. Graphical notations are powerful thinking and communication tools because they can convey a large amount of information at a glance. Harel [24] observes that a good graphical notation can improve thinking and believes that, "successful system development in the future will revolve around visual representations". Booch provides an informal diagramming notation called object diagrams for individual collaboration groups (Booch calls a collaboration group a mechanism). Unfortunately, the notation contains many programming-level details like object visibility and pointer passing.

The existing specification techniques that emphasize collaboration groups (e.g., Booch's mechanisms and Helm's contracts) treat each group as an independent entity. This is advantageous because the groups become abstractions that can be reasoned about separately from other concerns. The specification techniques do not address how the groups themselves are structured relative to one another nor how they operate in concert in a real system to achieve system-level behaviour. Elements of dynamic structure are not addressed, as is common to many object-oriented modelling techniques. Coleman observes that, "the design, efficient implementation, and documentation of dynamic systems can cause severe problems for object-oriented development teams" [18]. This may occur because, lacking appropriate tools, the developers are forced to reason about issues of dynamic structure in terms of programming language details like: pointers, and memory allocations and deallocations. Modelling techniques are needed which can express aspects of dynamic structure at the architectural level. For object-oriented systems, this means modelling the dynamic aspects of objects and collaboration groups.

In summary, there is a need for architectural models and notations that are appropriate

for representing collaboration groups in object-oriented systems. Existing notations do not adequately address all the needs of designers and framework users. Informal graphical notations are powerful because they can express the essence of a system's architecture. There is a need to experiment with the issues associated with the design of a real object-oriented system in terms of collaboration groups and dynamic structure.

1.2 Problem Statement

The problem of this thesis is how to model and represent the following aspects of an object-oriented system's architecture: (1) the architecture of individual collaboration groups, (2) the architecture of a system as composed of groups, (3) how the participants play their roles in multiple groups and the dependencies across groups, (4) the typical causal-flow patterns though the system, and (5) dynamic structure.

1.3 Thesis Objectives

An objective of this thesis is to develop an architectural model and supporting visual notation that is appropriate for expressing the architecture of object-oriented systems and frameworks in terms of collaboration groups. The resulting model is called the *role-based architectural model* (or just the *role-based model*) and it builds on existing material to the extent possible. The *role-based model* is inspired from the work by Helm on contracts [26] and includes inter-group dependencies and dynamic structure based on the model supported by the ObjecTime tool [45]. The visual notation is based on the MachineCharts notation of Buhr [11]. A subset of Buhr's Timethreads [10] are used to express the typical causal-flow patterns within a collaboration group and across collaboration groups.

Another objective of this thesis is to explore modelling issues associated with the role-based model by applying it to an existing object-oriented framework. Issues to be explored include: the use of causal-flow and timethreads to represent intra-group and inter-group behaviour patterns, the usefulness of representing a large portion of an existing framework in the role-based model, and the usefulness of representing inter-group dependencies and dynamic structure.

The case study in this thesis is of a conventional (i.e., sequential) object-oriented sys-

tem organized around a framework. One should not construe that this is the only type of system to which the techniques of this thesis may be applied. The thesis defers properties of the components (objects) and their connections (messages). In particular, there is no distinction made between active components that have a separate thread of control (e.g., processes) and passive components that do not (e.g., abstract data type modules). The properties of components is less of a concern when behaviour is expressed using causality-flow because causality-flow concerns the sequence of activities and not how that sequence is achieved. The role-based model and accompanying causality-flow specification in timethreads is an appropriate starting point for making decisions about active versus passive components and how control should be allocated. It is believed, therefore, that this work is applicable to systems that have active components, passive components, or some mix of the two.

The emphasis is on informal models and notations which focus on concepts and their representations with visual notations. It is believed that the concepts could be formalized, but the intent here is to use intuitive concepts and visually suggestive notations. The purpose is to develop models of important structural divisions and typical causal flow patterns as these form the essence of a design. Developing complete executable specifications is beyond the scope of this thesis. The models developed are of programs in operation; the thesis does not address issues associated with the organization of inheritance hierarchies. Design methods for using the proposed model are also outside the scope of the thesis.

1.4 A Research Overview

In response to these objectives, the following are the key developments arising from the research: a meta-model of architectural concepts for software systems, a role-based model, and an exploration of the issues associated with using the role-based model on a real system.

The Architectural Meta-Model

A meta-model is a model of model(s). The architectural meta-model is used to describe some architectural concepts that are applicable to a wide range of software and to motivate

a visual representation for these concepts. The meta-model addresses the following static aspects of architecture: the factoring of a system into collaboration groups, the structure of the groups, the dependencies among groups, and the interconnections among groups and group participants. The meta-model addresses the following dynamic aspects of architecture: the creation, destruction, and movement of components, and the temporal constraints on the operation of participants in groups. The concepts of the meta-model are organized into a specialization hierarchy that may be extended by refinement in an analogous way to subclassing in object-oriented class hierarchies.

The Role-Based Model

The *role-based model* is developed by extending the architectural meta-model. The role-based model introduces: *role teams* which are templates for a collaboration groups, *roles* which are templates for role team participants, *objects* which play roles, and *role methods* which perform activities in role teams. When one is considering the static structural organization of a system in terms of role teams, the teams and roles are acting together as a template for possible runtime organizations. When one is considering the operation of the system, perhaps by playing causal-flow patterns across it, the role teams are acting as instances (e.g., collaboration groups) and the roles as objects. Therefore, role teams and roles have a dual purpose in system models: they are *definitional* when they act as templates for runtime organizations, and they are *operational* when the behavioural scenarios through the system are played out. This duality of role teams and roles is analogous to the use of classes in many object-oriented design methods. For example, Rumbaugh [44] refers to his E-R diagramming notation as an object-model, even though the entities are classes and the relationships are drawn between classes.

This thesis takes a operational view of role-based models; thus, roles and role teams are considered operational components of a system's architecture. Helm [26] and Anderson [2] take a more definitional view and are considering how to construct class hierarchies from role-based models.

Exploring the Issues

The case study exercises the role-based model and its visual notation. A framework called HotDraw is reverse engineered into the role-based model. HotDraw is chosen because it is in the public domain and has been studied using other approaches [35]. HotDraw is a framework for developing semantic graphic editors. A sufficiently large portion of HotDraw was reverse-engineered to exercise the approach against the design of a framework as a whole.

1.5 Thesis Organization

Chapter 2 provides background information on object-oriented technology and frameworks. A review and critique of the relevant specification techniques for inter-object behaviour and collaboration groups is given. Chapter 3 presents the meta-model and its visual notation. Chapter 4 extends the meta-model introduced in Chapter 3. The result is the role-based model. A small example that applies the model is presented. Chapter 5 presents a case study. Chapter 6 offers a discussion and presents avenues for future work.

Chapter 2: Basic Principles and State of the Art in Object-Oriented Modelling

The basic principles of object-orientation as characterized by terms like *objects*, *classes*, *inheritance*, *polymorphism* and *message passing* are discussed. Object-oriented frameworks are described by examining an extension to an existing framework. A review and critique of object-oriented modelling techniques is given. Many of the diagrams used in this chapter are meant to convey concepts and do not represent notations developed as part of this thesis.

2.1 Basic Object-Oriented Principles

An *architectural view* of object-oriented systems may be contrasted with a *constructive view* [13]. An *architectural view* emphasizes the operational components that form a designer's conceptual model of a system in execution and how they interact, i.e., how the system looks and works as a running system. Objects, subsystems, and processes are examples of operational components; roles and role teams as used in thesis are also operational components. In contrast, the *constructive view* emphasizes how the operational components are achieved via the building materials of object-oriented technology. For frameworks, this view would include classes and the organization of inheritance hierarchies.

The two views are not orthogonal and both are important aspects of design. Here they are used to provide reference points within which object-oriented systems can be studied. This section begins with a basic architectural view of object-oriented systems and then demonstrates how the building materials are used to achieve it.

2.1.1 Basic Architectural View (Object Architecture)

The basic architectural view, or more simply the object architecture, of object-oriented programs is that of *objects* communicating through *message passing*. The top half of Figure 1 shows the elements of an object architecture in visual form. *Objects* have identity and encapsulate both state and behaviour. They are the basic building blocks of an object-oriented system. The state of an object is contained in *instance variables* which may reference other objects to create a directed graph of objects. The behaviour of an object is contained in *methods* which operate on the instance variables. Methods may be categorized as *public*, meaning they are available to client objects or *private*, meaning they are unavailable to clients but are necessary for achieving the behaviour of the public methods. Methods may also be categorized based on how they are used in the constructive view (Section 2.1.2).

A message is a request from a client object to a server object to perform one of its public methods. Prior to sending a message a client must obtain the identifier (object reference) of a server and supply that identifier as part of the message. An object graph with objects as the nodes and object references as the arcs tends to be dynamic as object identifiers are exchanged to support message passing. Obtaining an identifier to an object is equivalent to dynamically establishing a connection for communicating to the object that the identifier names. Servicing a message may involve many objects, perhaps requiring new objects and object references to be established dynamically, and the objects involved may, through time, act as clients, servers or both.

The bottom half of Figure 1 is a series of snapshots that show a portion of an object graph through time. The arcs from inside one object to the edge of another are object references from the object at the tail to the object at the head. The second frame shows that new object references may be created; this is achieved by the exchange of identifiers over existing connections. The new connection from B has been dynamically-bound to A and B may now send messages to A. The third frame shows the creation of a new object and a reference to it, and the destruction of an existing object and its associated references. Dynamic object graphs, like the shown here, seem typical of object-oriented programs [13].

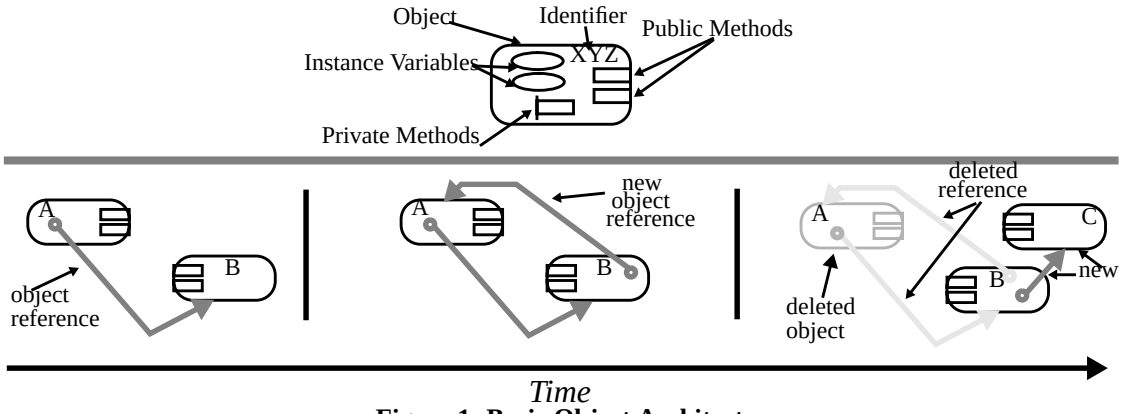

Figure 1: Basic Object Architecture

The arcs between objects in Figure 1 are references to objects (i.e., object identifiers). These arcs are not the same, necessarily, as the communication pathways between components in more abstract architectural models of systems. Object references are viewed as a mechanism for achieving communication pathways. For example, one may model a communication pathway between objects as static in a more abstract architectural model, when in fact object identifiers are exchanged dynamically to establish the pathway or each time communication across it is required. As another example, a communication pathway may be modelled as direct when the actual message path is routed through a third component.

Message passing is the only way that one object can interact with another. A message specifies *what* a server must do and not how the server must do it. The methods of a server specify *how* a message is serviced and are the only way (ideally at least) that the private state information of an object may be changed. Goldberg [22] observes that these properties increase system modularity, which has positive side-effects like maintainability and reusability.

The dark-side is that few object-oriented programming languages place restrictions on the availability of object identifiers and what may be done with them in different contexts. For example, programmers can violate object encapsulation by making an object's state information available to other objects through exposed references to that state information. The result can be complex object graphs (like the one shown in Figure 1) with access paths from multiple places to an object, and/or multiple paths from the same place to an object.

Multiple access paths to an object indicate the possible “roles” that the object plays at a point in time. This can create problems if the aliased object is asked to play roles that are inconsistent [27], e.g., to be a source and a destination of a matrix multiplication operation. Hogg [28] observes that the complexity introduced by aliasing can make understanding object-oriented systems difficult and he presents language features to manage the problem. Role-based modelling can make potential aliasing explicit during design by describing an object in terms of the roles it plays in relation to other objects.

2.1.2 Basic Constructive View

Object-oriented programs are not assembled by configuring objects and object references, but by organizing: classes, polymorphism, and inheritance. These must be organized to achieve the desired runtime architecture and to fulfil design goals like reusability and extensibility.

Classes are the physical building material that object-oriented programmers use to construct systems. Classes can be categorized as abstract or concrete. An *abstract class* is a template for creating other classes. An abstract class has at least one method that is incomplete, so it would be an error to create an instance of an object from an abstract class (although there is nothing in most object-oriented languages to stop this). A *concrete class* has complete implementations for all of its methods, making it a template for creating object instances.

Polymorphism is the ability of a single variable or procedure reference to take on different values from different types. In object-oriented programs, the actual target method of a message send is based on the class of object that is receiving the message. With polymorphism there may be a choice of which target method is used based on the type (class) of a variable that references an object. If a polymorphic reference can take on different values from different types dynamically, then polymorphism is achieved by *dynamic binding*. In most object-oriented languages this means a table lookup for target methods at runtime. Without dynamic binding, polymorphism is just a syntactic convenience that is equivalent to overloading in languages like Ada83.

Polymorphism allows for the development of generic behaviours that are implemented using polymorphic message sends. The generic behaviour can be reused without change when new classes are added provided the new classes provide the polymorphic methods (meaning the target method that is called in a polymorphic way). An important side-effect of polymorphism is that it encourages consistency in naming and behaviour across the system, because equivalent methods in different objects will have the same signature, and objects used in related ways will have similar interfaces [37].

Inheritance enables classes to be organized into generalization/specialization trees with abstract classes near the root and more specialized concrete classes near the leaves. As a modelling technique, inheritance is useful for formalizing the “is-a” relationships between classes and is used in many object-oriented requirements analysis techniques [17][44][55]. As a reuse technique, inheritance allows common behaviour to be located in a central place from which it can be easily reused through subclassing. A *subclass* is a more specialized version of its *superclass* from which it is a decendent. A subclass automatically inherits the behaviour of its superclass.

2.1.3 Example: Use of Object-Oriented Construction Materials

The example of Figure 2 illustrates how the object-oriented construction materials are used in combination to create flexible and extensible software. The figure illustrates an inheritance tree using the notation of Rumbaugh [44]. (A summary of Rumbaugh’s notation appears in Appendix B.) Classes are shown as boxes; an inheritance relationship is shown as an arc annotated with a triangle; and methods are shown as text strings inside the class boxes. This inheritance tree represents a portion of the classes found in the HotDraw framework [35] for creating semantic graphic editors for 2D geometric figures.

The classes shown in Figure 2 model the graphic objects that users manipulate on a computer screen. The **Figure** class is the abstract superclass of the **CompositeFigure, PolylineFigure**, and **TextFigure** concrete subclasses.The **Figure** class has three methods **displayBox**, **origin**, and **extent** (many of the methods have been omitted for simplicity).The **displayBox** method returns a rectangular area that represents the region of the screen on which a figure is drawn. This rectangular area is used to update the display after

the display has been damaged by an action such as dragging a figure.The **origin** method returns a point representing the upper left corner of a figure's rectangular display area, and the **extent** method returns a point that represents the width and height of a figure's rectangular display area. The calculation of the origin and extent is dependent on the specific subclass of **Figure**. For example, a **PolylineFigure** would calculate the origin and extent based on a collection of points that define the vertices of a polyline, and a **TextFigure** would calculate the origin and extent based on the type of font used and its size. The **origin** and **extent** methods must be *deferred* methods of the class **Figure**, meaning that the implementation of the methods is the responsibility of a subclass. The **displayBox** method can be implemented in the **Figure** class using the points returned by the **origin** and **extent** methods to construct and return a rectangle of the proper size and position. The **displayBox** method is referred to as a *generic* method because it uses methods that are deferred. Viewed from the concrete classes, the **origin** and **extent** methods are *base* methods because they have complete implementations.

The **displayBox** method represents a generic behaviour that is reused across all subclasses of **Figure**. New subclasses can be added, e.g., semicircles, without changes to the existing system provided the new subclasses provide the **origin** and **extent** methods. All the methods shown can be called in a polymorphic way by clients of the **Figure** classes. This promotes reusability of client code and encourages common interfaces among related classes. The combination of inheritance, polymorphism, and method usage illustrated by this example is a common way of organizing object-oriented construction materials to achieve reusability and flexibility in frameworks.

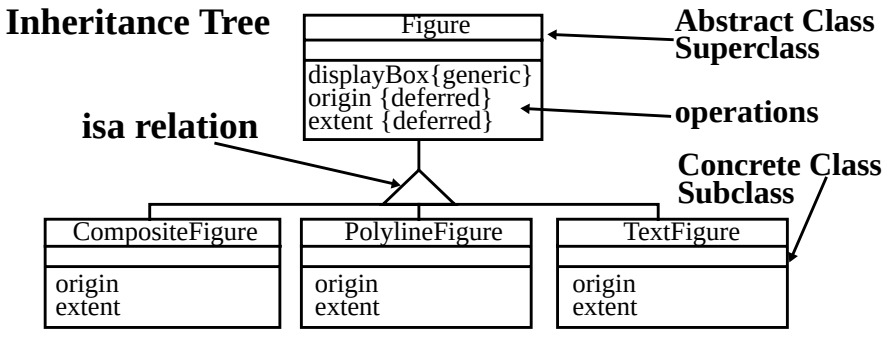

Figure 2: Inheritance, Classes, Polymorphism and Methods

2.1.4 Object-Oriented Programming Languages

Object-oriented programming languages offer varying levels of support for the concepts discussed in this section. Smalltalk has only public methods, the distinction between public and private methods is managed by programming conventions; and provides no pre-runtime support for checking for the existence of necessary deferred methods. It is possible, however, for the programmer to specify that a deferred method must be provided by sending the message **subClassResponsibility** from the superclass when the deferred method is called, and the system will report an appropriate error message. C++ [20], a strongly typed object-oriented language, has more support for the abstractions discussed in this section. C++ has support for public and private methods, and distinguishes between methods that are provided for clients versus those provided for subclasses. A distinction is also made between polymorphic methods (virtual functions) and non-polymorphic methods.

2.1.5 Mapping between the Basic Views

When developing object-oriented programs it is necessary to understand the object architecture, the constructive view, and the relationships between them. The object-oriented building materials (i.e., inheritance, classes and polymorphism) result in a factorization of a system into reusable elements. From an architectural standpoint, much of this factoring can be aggregated to create a simpler model of the program in operation. Figure 3 illustrates how paths through an inheritance tree may be collapsed to create collapsed leaf objects. The collapsed leaf objects are operational components in a object architecture. Conceptually each leaf object is given its own copy of all the properties (methods and instance variables) that occur along the path in the inheritance tree from its concrete class to its most abstract superclass. Object-oriented runtime systems achieve this same effect by automatically propagating messages sent to object instances up the inheritance tree in search of matching methods.

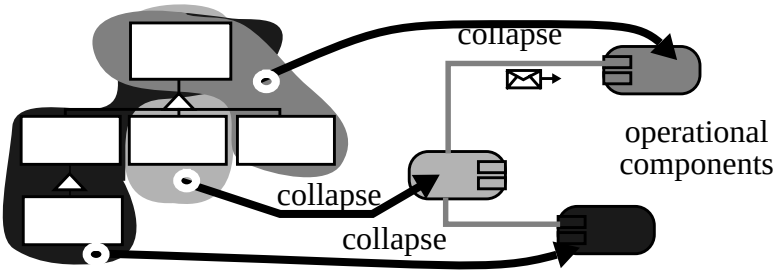

Figure 3: Collapsing Inheritance Trees to achieve an Architectural View

2.2 Basics of Object-Oriented Frameworks

A framework is constructed as a hierarchy of related classes that may be reused by extension to create new systems in a given problem domain. Frameworks are powerful because they embody the behaviour patterns that join objects into collaboration groups. These behaviour patterns are implicitly reused when subclassing off a framework. This is in contrast to the reuse of a software library which reuses only the functionality of the library and not the control code needed to manipulate it. The use of frameworks has been termed design-reuse by Deutsch [19] because of this distinction and Johnson [35] defines a framework as abstract design for a particular kind of application. This section discusses some basic properties of frameworks by examining the HotDraw framework and by applying a simple extension to it.

HotDraw is an object-oriented framework for creating semantic graphic editors for 2D drawings. It can be used to build editors for specialized drawings, such as hardware schematics or software design diagrams. Figure 4 shows the user interface of HotDraw before it has been customized; it is a complete and useable application before being extended. Figures are drawn on a drawing canvas by selecting an appropriate tool from a tool palette. The initial set of figures include: lines, arrows, rectangles, circles, and text. Standard operations on the figures include: resizing, dragging, erasing, grouping, and saving.

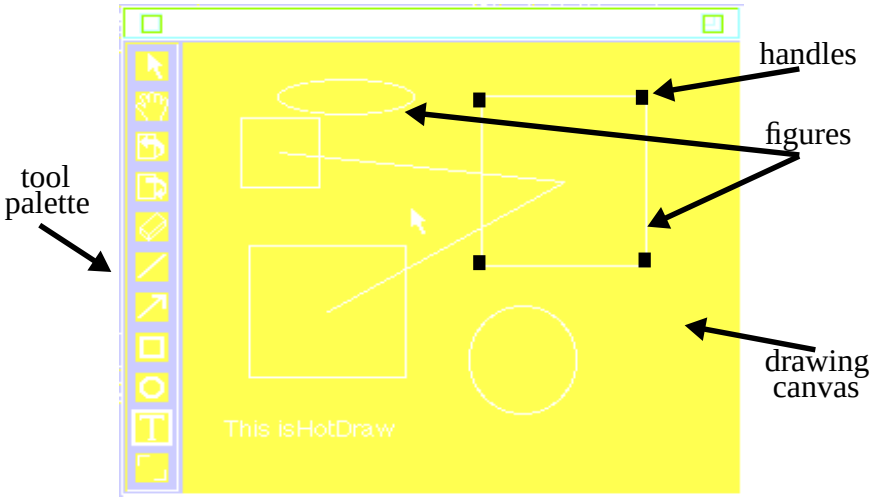

Figure 4: HotDraw A Semantic Editor for 2D Graphics

Users of HotDraw can manipulate figures in several ways. Direct interaction is achieved by selecting a figure with the selection tool and then manipulating the figure's handles. Handles appear on figures as filled squares. Handles may be used to change a figure's size or shape. It is also possible to pop up a menu of operations to perform on a figure, e.g., to change a figure's colour. A useful feature of HotDraw is its ability to maintain constraints between figures. One such constraint allows for arcs to be connected to other figures such that when the figure moves the arcs moves as well. This is a common requirement for many semantic graphic editors.

Figure 5 is a constructive view of the HotDraw class hierarchies. The organization and naming of the classes and class categories lets us infer much about HotDraw: **Figures** classes represent the figure objects that are drawn with the tool, **Tools** classes represent the objects that appear in the tool palette and manipulate the figures, **Constraints** classes represent the objects that enable constraint management, **Handles** classes represent the handle objects that appear on figures, **Canvas Control** classes represent objects that manage user interaction in the drawing area, and **Tool Palette Control** classes represent the objects that manage user interaction in the tool palette area.

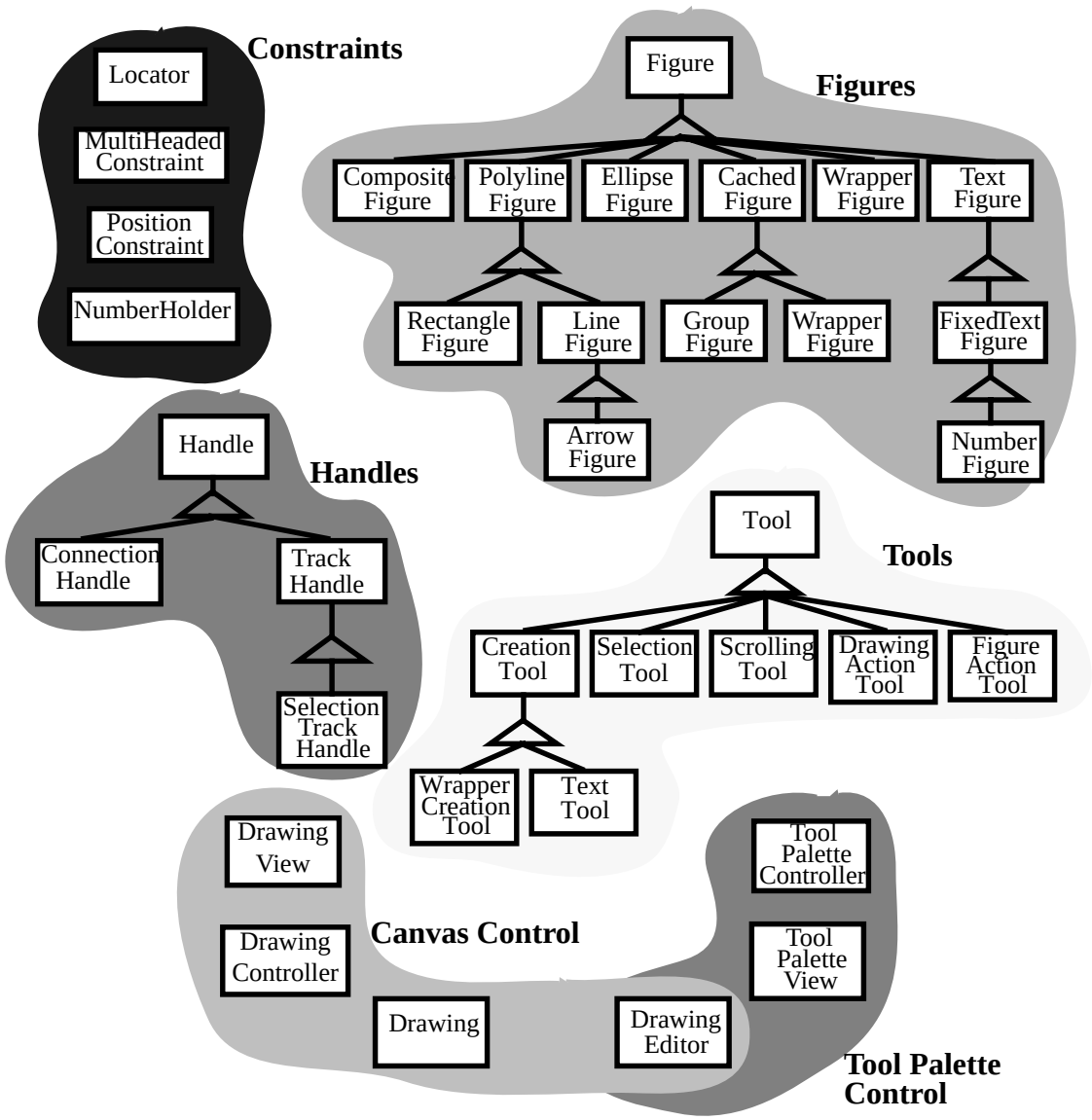

Figure 5: The Class Hierarchies and Class Categories of HotDraw

What Figure 5 does not explicitly represent are the behaviours that span the classes in the inheritance trees and the collaboration groups that the object instances participate in. For example, there must be interaction patterns between **Handles** and **Figures** that enable manipulations of a **Handle** to effect the properties of a **Figure**. Likewise, there must be interactions between the **Constraints** and the **Figures** that allow dependencies between **Figures** to be maintained. Other interaction patterns are imaginable, like those between objects

of the **Canvas Control** classes to control user interactions. If the interaction patterns and the collaboration groups they imply are treated as entities, then one can imagine that the collaboration groups must cooperate and be linked through time to create a running system.

To illustrate the power available through the reuse of frameworks, the HotDraw framework is extended to include triangle figures. This may lead to the class extensions shown in Figure 6. Judging from the names of the classes and their organization, it would seem logical to add a subclass called **TriangleFigure** under the **PolylineFigure** class. Using the **RectangleFigure** class as a guide one can determine the methods and behaviour required of the new **TriangleFigure** class. Optionally one could consult documentation for the framework, like Johnson's patterns [35], to discover what code is needed in the new **TriangleFigure** class. The triangles added to HotDraw are constrained to be right angle triangles to simplify the interaction with the user when the triangles are added (otherwise the user must specify three vertices instead of only two). The **TriangleFigure** class consists of one page of code (including comments), 11 methods of which 8 are identical to **RectangleFigure**, and 7 new or modified lines of code across the remaining 3 methods. Another subclass is also needed to add a new tool to the tool palette to allow the creation of triangle figures. This new class has one new method with 2 lines of code.

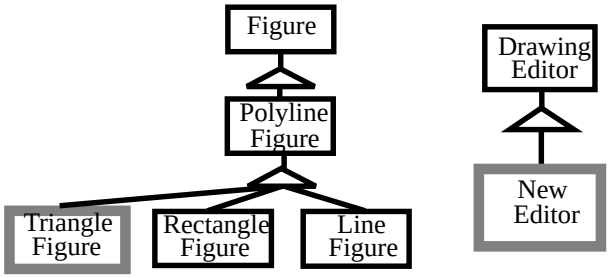

Figure 6: Extending HotDraw with Triangular Figures

The key point of the above example is the amount of behaviour that is reused relative to the effort spent to perform the extension. **TriangleFigures** like other figures may be selected, resized, moved, shuffled, saved to disk, copied, deleted, etc. Some of the inherited behaviour is gained without **TriangleFigure** providing any behaviour of its own: the necessary behaviour is inherited from abstract superclasses. In other cases, however, the new subclass is expected to provide methods that are deferred by its superclasses and that are

invoked by the other objects of the framework.

The example illustrates the flow-of-control in frameworks. The **TriangleFigure** class must provide the methods deferred by its superclasses, as well as methods required of it given the behaviours that its superclasses involve it in. These methods will be activated along control flow paths that originate in the framework. This represents an inversion of control relative to subroutine libraries where the added application code is responsible for driving the flow-of-control. The advantage of the framework approach is that the design of the flow-of-control is reused across applications. Reuse of the flow-of-control implies that interface design and the division of responsibilities between the components of a system are also reused. After studying the frameworks of the Smalltalk-80 system, Deutsch [19] remarks that interface design and the division of responsibilities constitute the key intellectual content of software and that they are far more difficult to create or recreate than code.[1]

The chapter now begins a review and critique of the common approaches used by object-oriented design methods and tools to model object-oriented programs and frameworks.

2.3 Review and Critique: E-R Modelling

An approach taken by many object-oriented analysis and design methods [44][17][55] is to extend inheritance relationship diagrams with concepts from Entity-Relationship (E-R) models [16]. Classes become the entities of the E-R model and inheritance becomes but one of the relationships between the entities, other relationships include aggregation (part-of) and general association. *Aggregation* allows one to model a larger entity as a composition of parts, and *association* is used to model general relationships between entities.

An E-R model captures the static structure of a system by showing the classes in the system, the relationships between the classes, and the attributes and operations that each class supports. The model typically simulates the entities and entity relationships that can be derived from the problem statement and problem domain. E-R modelling is often advocated because it offers a consistent paradigm from requirements analysis to implementation

[1] Deutsch uses the phase *functional factoring* where the phase *division of responsibilities* is used here. We believe the two to be equivalent.

that aligns well with the basic construction materials of object-oriented programming. It is believed that such models result in systems that are more resilient to change [44] than do models which partition a system based on the functions it performs (e.g., structured analysis methods).

Figure 7 extends the example of Figure 2 using Rumbaugh's E-R modelling notation (a complete summary of Rumbaugh's notation appears in Appendix B). The model specifies that many **Figure** objects are contained in a **Drawing** object (aggregation), and that a **Drawing** object is *displayed on* a **DrawingView** object (association). The small arrow next to a label (e.g., the *displayed on* label) specifies the direction to be used when reading a label. The filled circles at the end of a relationship represent multiplicity of the object instances involved in the relationship; e.g., many **Figure** objects are contained in a **CompositeFigure** object. This is an example of a recursive aggregation relationship; it allows for nested figures.

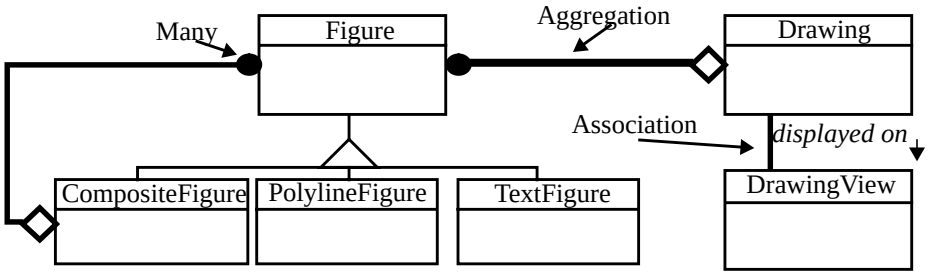

Figure 7: Extending the Basic Constructive View

Although an E-R model, like the one in Figure 7, is expressed using classes, it is really a template for many potential runtime organizations of object instances. The classes of an E-R diagram represent construction templates for objects, and the relationships between classes represent potential runtime relationships between object instances. For example, aggregation is described as a relationship between classes, this does not mean that classes are composed of other classes, it means that the *instances* of classes are composed of the *instances* of other classes. This illustrates the dual purpose of E-R models of classes: the "is-a" relationship between classes is a constructive relationship that describes how objects are built from classes, the aggregation and association relationships have architectural im-

plications because they describe the organization of a system in terms of objects (operational components).

Figure 8 shows how the E-R model of Figure 7 may be interpreted in the basic object architecture. The **DrawingView** object and the **Drawing** object each have an object reference to the other. These references are used to direct messages related to the *displayed on* relationship in the E-R model. **Drawing** objects reference **Figure** objects to achieve the part-of relationship between Drawings and Figures in the E-R model. The **CompositeFigure** object has object references to **Figure** objects contained within it. Figure 7 represents one possible way of achieving the relationships of the E-R model in terms of the object architecture. There are other ways to achieve the E-R model (see Rumbaugh [44]), for example, the part-of relationship could be implemented using a separate object for the aggregation relationship and the **Drawing** object would interact with the aggregation object as needed.

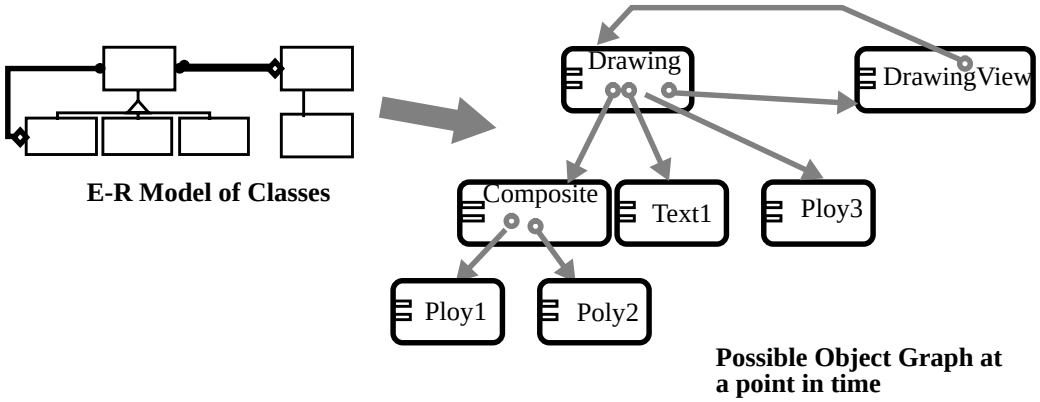

Figure 8: E-R Model Defines Topology of Object Graph

An E-R model represents the topology of one instance of a directed graph of objects. For example, if there were multiple **Drawing** and **DrawingView** objects, there would be multiple object graphs like the one shown in Figure 8 occurring at runtime.

2.3.1 Critique: E-R Modelling

The benefits of the E-R model extensions are: (1) a high-level consistent paradigm that maps well to object-oriented languages, (2) clues to behaviour in terms of the relationships

between objects and the responsibilities of those objects in the relationships, and (3) a description of the topology of runtime object instances. An E-R model, however, is static and lacks the descriptive power necessary to model important aspects of system behaviour, such as global causal-flow patterns and local causal-flow patterns of collaboration groups.

2.4 Review and Critique: State Machine Modelling

Several authors have added a complimentary behavioural view to the E-R model approach. The most common technique uses state machines to model the internal behaviour of object instances. Techniques and tools that take this approach include the ObjecTime tool [45], Rumbaugh's OMT method [44], Shlaer and Mellor's Object Lifecycles method [46], and the ObjectCharts approach of Coleman et. al. [18]. Other behaviour description techniques for objects have been used and are similar to the state machine approaches: the G++ toolset uses the State Description Language (SDL) [42], PROTOB uses a variation of Petri Nets [5], and Jackson's Structured Design [34] uses a graphic representation of regular expressions called tree diagrams.

Figure 9 shows the two most common models for object-oriented development: E-R Models and communicating state machines. Regardless of the detailed differences between the approaches the basic operational mechanisms are the same. Each object is modelled as having an internal state machine. The machines are driven by events caused by messages sent between objects. Objects perform actions during state transitions or (optionally) while in a state. Actions may be local to an object or include messages to other objects. State machines are usually expressed using some variation of Harel's statecharts notation [23]. The semantics of state machine modelling in most object-oriented techniques are not formally defined; however, there have been attempts to formalize the approach [25].

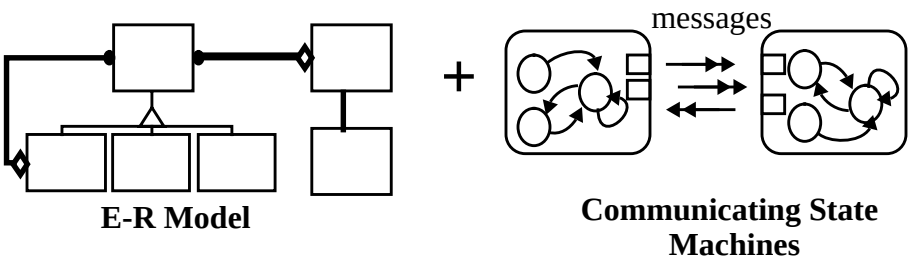

Figure 9: Common Models for Object-Oriented Design

The different techniques and tools that use the communicating state machine approach differ in their detailed semantics. For example, Rumbaugh and ObjecTime support inheritance of state machines, but Shlaer and Mellor do not. Communication between state machines in the ObjecTime tool is point-to-point and may be synchronous or asynchronous. Events generated in Rumbaugh's model can be directed to a specific object, or to any and all objects that accept the event; however, the semantics of the communication primitives (e.g., synchronous, asynchronous, etc.) are not defined. ObjecTime, Rumbaugh, and ObjectCharts all use Harel'style state machines, but Shlaer and Mellor use conventional flat state machines. State machine actions are specified in ObjecTime in a conventional object-oriented programming language, Rumbaugh uses structured text that is linked by naming convention to the operations supported by classes in an E-R model, and ObjectCharts uses a formal specification technique that is a derivative of VDM [38].

2.4.1 Critique: State Machine Models

Although the state machine approach is powerful, it has limitations for expressing the aggregate behaviour of groups of objects. State model diagrams obscure the interactions between objects by hiding them in the actions taken during state transitions or while in states. Notations that use ill-defined communication semantics (e.g., Rumbaugh) worsen the problem because the "overall system behaviour cannot be deduced from the behaviour of individual objects" [18]. Precise communication semantics as in ObjecTime can help, but one must often examine code fragments to understand the point-to-point communication primitive used. Also the "every object has a state machine" approach may lead to a very fine-grained and fragmented behaviour specification. These limitations may make it difficult to

piece together the aggregate behaviour of a group objects and to understand the typical behaviour patterns among objects.

Another issue concerns the detailed closed-form nature of state machines and its effect on system modelling. First, it may involve too much effort to specify many objects in a system in state machine form and may even be counter-productive because relatively few objects have rich enough behaviour to warrant a state machine description. Second, as a specification tool, an exclusively state machine approach may cause designers to commit to too much detail during the early stages of design [11].

2.5 Review and Critique: Models for Collaboration Groups

The basic constructive view suffered from its lack of behavioural characteristics, but the basic object architecture is perhaps too dynamic and low-level to be useful as a modelling tool (although it is a model that object-oriented programmers must deal with in practice). Building on the basic architectural view, it is possible to add new abstractions, both structural and behavioural, to deal with these problems.

An obvious approach is to introduce larger grained structures by grouping related objects. Several authors have identified the need for *subsystems* [8][44][53]. From an architectural perspective, a *subsystem* is a collection of logically related objects (and possibly other subsystems) that are participating to provide the services required of a major functional division of a system. Examples of the functions performed by subsystems include: window management, database management, process scheduling, and error reporting.

Subsystems are very large and objects tend to be very small, so there is a need for medium grained abstractions to bridge the gap. Independently, researchers have developed notations for specifying the organization of small groups of objects that are collaborating to achieve some objective, such groups are referred to as collaboration groups.

Booch defines a *mechanism* as any structure whereby objects work together to provide some behaviour that satisfies a requirement of a problem [8]. Booch has developed a graph-

ical notation called *object diagrams* for specifying the structure of mechanisms. The elements of the notation include objects and their interconnections. Each object diagram represents one possible instantiation of a mechanism. Booch's object diagrams also include notation for specifying details of object visibility. For example, object **A** may have an instance variable reference to object **B**, or **A** may be given the identifier of **B** as a parameter of a message send. Object references may be further categorized based on how they are used, e.g., **A** may use the reference to **B** in its public methods or **A** may use the reference to **B** in its private methods. The result, is a notation that mixes programming-level concerns (like the details of gaining object visibility) with architectural concerns (like the operation of the mechanism).

Helm et. al. [26] organize an object-oriented system into groups of interacting objects called *contracts* and provide a textual formalism for specifying individual contracts. A *contract* is a specification of how groups of interdependent objects participate to accomplish tasks. A contract is composed of a set of contract participants, any invariants maintained by the contract, and a specification of how the contract is instantiated. A *contract participant* is a formal specification of the structure and behaviour of objects that may join the contract. The structure of participants are defined by *type obligations*, i.e., a set of interface methods and internal data attributes. Holland [31] has extended the contract notation to include private methods for contract participants. The behaviour of participants are defined by *causal obligations*. A causal obligation is a formal algebraic notation that expresses in a procedural sense the activities that the methods of participants must perform. The activities of methods may include updating local data attributes and the sending of messages to other contract participants. Contracts may be nested recursively into subcontracts and organized into refinement hierarchies. Contract refinement is similar to class inheritance except that contract refinement it is applied to a group of objects.

A contract participant is more general than an object. An object is derived from a class which is a construction template that represents the complete aggregate structure and behaviour of an object. A contract participant, however, represents only the structure and behaviour of an aspect of an object necessary for that object to participate in a contract. A contract specification is more abstract than a mechanism, and therefore, potentially more

reusable in different systems. The distinction between contract participants and classes requires that a contract specification be mapped to classes in inheritance hierarchies. Helm provides a formalism called conformance declarations that can define the mapping.

Arapis [3] presents a model of object behaviour and object cooperation based on objects, messages, roles, and contexts. A *composite context* is composed of a collection of components (objects) and a collection of component constraints. A composite context defines the cooperation among a set of components as an interleaving of roles that are played by the components as a group. A *role* represents an aspect of behaviour that a group of components exhibit. A role has a set of methods that are sent and received by the composite context and rules for their ordering. A role has a set of methods that are exchanged between the composite context and its internal components. The components of a composite context may only exchange messages with the context and not among themselves. Public constraints associated with a composite context define the temporal order on the interleaving of roles; public constraints are specified using predicate temporal logic. A composite context is analogous to Helm's contracts: components are equivalent to contract participants, component messages equate partially to type obligations, and the rules governing the ordering of messages are like causal obligations. The components of a composite context, however, may not communicate with one another. This does not reflect how object-oriented systems are built in practice.

ObjecTime [45] is a computer-aided software engineering tool for developing object-oriented real-time systems. Systems are modelled using black-boxes called *actors* that communicate over *bindings*. A binding is a fixed connection between two actors over which signals between the actors flow. Internally an actor may be composed of other actors and may have an internal behaviour expressed as a state machine. An actor is a black-box that may only communicate through messages to its surrounding environment. All actors are modelled as concurrent entities that communicate by message passing. The ability to decomposed actors means that actors can be used to model collaboration groups. The collaboration groups are actors that have internal structure, an interface, and internal behaviour. Actors can be organized into inheritance hierarchies that can separately refine the structure and the behaviour of a decomposed actor. This is analogous to the contract refinement ca-

pability of Helm.

ObjecTime has some modelling features that can express the structural and temporal constraints across collaboration groups (actors). An individual actor may be part of several other actors, perhaps simultaneously, through the use multiple part-of and equivalence. Each decomposition in which an actor participates, defines one aspect of its total structure and behaviour. An equivalence relationship defines all the decompositions in which an individual actor may participate. The temporal properties of actors can be expressed as *fixed*, *optional*, or *imported*. Fixed actors are created implicitly when their containing actor is created, optional actors are created dynamically at runtime, and imported actors model cases where an actor is imported from elsewhere to operate temporarily in a given decomposition. Actors may be dynamically created and destroyed. Models built using the ObjecTime tool are executable.

Johnson [35] uses a structured essay approach to document frameworks for the users of them. A pattern specification implicitly describes the interaction patterns between groups of objects in a framework that users of a framework must understand when extending it. The framework is documented as a set of "patterns". A pattern is a typical use of framework. The first pattern describes the overall purpose of the framework and gives examples of how it can be used. Successive patterns describe progressively more detailed uses of the framework. For example, the first pattern in an essay might describe a framework as a generic architecture for constructing graphical editors. Subsequent patterns might describe how to extend the menu palette to include new graphical figures or how to render the graphical figures on a drawing canvas. Each pattern begins with a problem statement that describes the purpose of the pattern. A description of the pattern then follows. This is a free form discussion that often describes the interaction patterns between groups of objects by giving concrete examples. The pattern concludes with a summary of how to extend the framework in terms of what classes and methods are needed to achieve the pattern. Links to other related patterns are also given.

2.5.1 Critique: Models for Collaboration Groups

The techniques of this section move the focus from being primarily on the behaviour of an

individual object to the interactions of a group. These techniques are better suited to the requirements of a specification technique for collaborating objects and frameworks than techniques for specifying individual objects.

Formal-textual techniques like Helm's contracts and Arapis's contexts are potentially powerful. For example, Arapis develops a satisfiability algorithm for verifying the consistency of the specification expressed in his formalism, and Helm et. al. are developing language support and tools based on their formalism. Unfortunately, as tools for system understanding and design these techniques may be difficult to use. They embed object interactions in operation specifications that are not centralized making it difficult to determine the behaviour patterns through a collaboration group. Also, textually-based approaches cannot convey the essence of a design at a glance as graphically-based approaches can, and their formal nature may lead to a level of completeness that makes the essence of system's architecture difficult to extract.

Booch provides graphical notations for mechanisms. Unfortunately the diagramming notation used by Booch emphasizes detailed programming level issues, such as object visibility and pointer passing. As a communication tool these details may obscure the essence of a design, and as a design tool they may cause a shift in focus to programming level concerns before the design issues have been resolved.

ObjecTime also provides a graphical notation; unfortunately the graphical notation says very little about the underlying behaviour of a system which is expressed textually in the code fragments associated with state machines. It thus becomes difficult to understand the relationships between the parts of a system and to piece together the aggregate behaviours that span the parts. Unlike the other models, ObjecTime can express elements of dynamic structure (e.g., creation, destruction, and movement of components) and the structural and temporal constraints across collaboration groups, albeit non-graphically.

Johnson provides a structured essay approach that describes how a framework may be extended in standard ways. The approach describes the design of a framework implicitly. Therefore, when novel framework extensions are attempted the pattern specification alone does not provide enough information. One must rely on reading the source code or on an-

other specification technique.

2.6 Review and Critique: Aggregate Behaviour Models

Describing the structure of larger components is only half of the picture; the internal behaviour of the larger components and the behaviours that span them are also important.

Booch uses *timing diagrams* to describe the flow-of-control within a mechanism. Figure 10 illustrates the style of Booch's timing diagrams, which are typical of the behaviour specification techniques in several object-oriented methods, e.g., Jacobson's interaction diagrams [31]. Each object in a mechanism is given a horizontal axis, time proceeds from left to right, and messages sent are shown as arcs between the object axes.

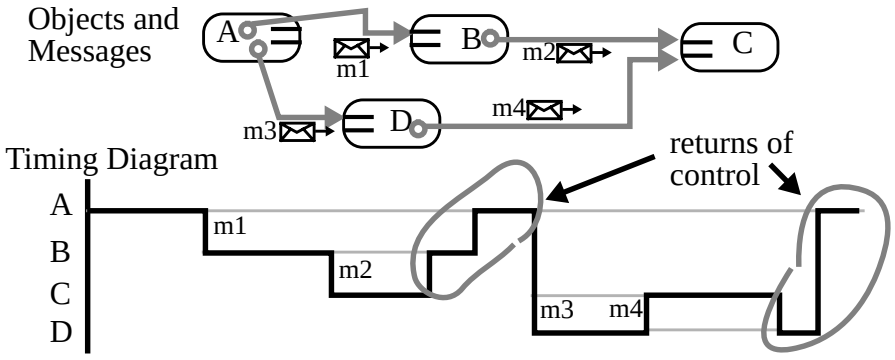

Figure 10: Typical Behaviour Representations of Object-Oriented Methods

As used by Booch, timing diagrams represent the explicit flow-of-control as it occurs in a running system. Timing diagrams represent issues such as, who calls whom, the nesting of calls, and explicit returns of control (see Figure 10). The grey-horizontal lines in the timing diagram of Figure 10 represent the nesting of calls. After message **m2** has been sent, for example, the call nesting level is two deep. The highlighted parts of the timeline represent explicit returns of control as the call stack is unwound.

Jacobson [32] defines a *use case* as a sequence of transactions, performed by a user and a system in dialogue. A transaction is performed by either the user or the system and is initiated by a stimulus. A use case is a textual, scenario style description that enumerates se-

quences of observable events that occur at the edges of a system. A use-case describes the operation of a system from a user's perspective: the system is treated as black-box that can be described from its observable behaviours and the system's internal components abstracted away.

Message sequence diagrams describe the sequencing of messages between objects. Jacobson has a variation called *interaction diagrams* that describe the mapping of a use case to the system's internal components. Rumbaugh's event traces [44] are another variation. Figure 11 represents a generic message sequence diagram for the example of Figure 10. Vertical lines represent the lifetimes of objects and directed arcs between the vertical lines represent message sends. The source of the arc is the client of the message and the object at the end of the arc is the server. Time proceeds from top to bottom. Message sequence diagrams are similar to Booch's timing diagrams because they explicitly represent client/server message sending relationships, but returns of control are not shown.

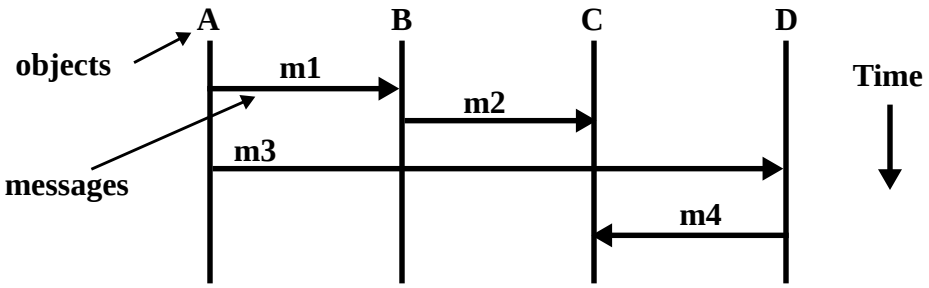

Figure 11: Message Sequence Diagrams

Buhr introduces *timethreads* [13] as a large-scale behaviour description technique for causality-flow. *Causality-flow* is a time-ordered sequence of causality connected activities performed by a system in relation to its architecture. A *timethread* is a causality-flow diagram. A timethread starts at some point of stimulus, touches elements of the design in the order they are activated and eventually terminates with completion of processing, or the delivery of a response. Time increases from the beginning of a thread to its end. Timethreads have a visual representation that looks like a curve and are often drawn over some representation of a design to show the relationship between causal-flow paths through the system and the architecture of system [13]. The complete notation can represent issues like:

branches and joins in the thread path, the creation and destruction of components, waiting patterns, and complex interaction patterns between concurrent timethreads [14]. A summary of the timethread notation that is used in this thesis is given in Appendix A.

The example of Figure 10 is repeated in Figure 12 using timethreads. A timethread is overlaid on the object diagram in the top part of the figure. The filled circle represents the start of the thread and the thick-straight line represents its end-point. The path between the start and end-point links the objects as they are activated through time (e.g., **A**, **B**, **C**, **D**, **C**). In the bottom half of the figure is a timing diagram drawn using a timethread interpretation. Note that the timethread expresses the pure sequencing of activities; detailed control-flow issues, such as returns of control, are not necessarily represented.

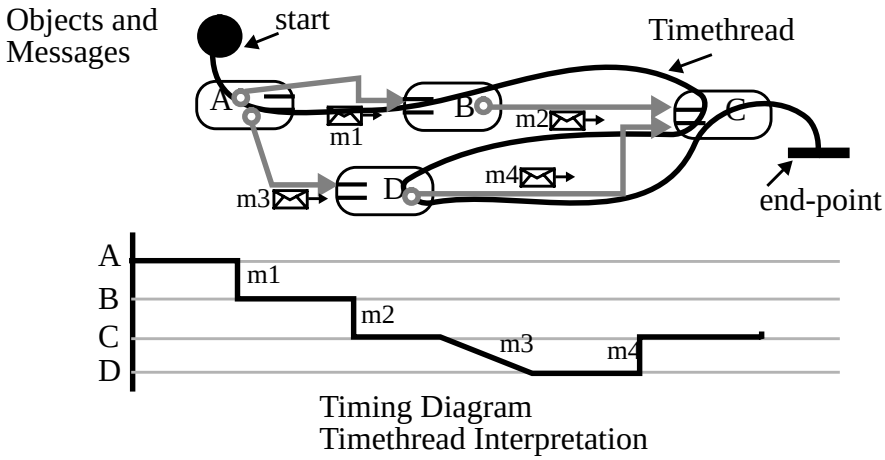

Figure 12: The Timethread Representation of Behaviour

Timethreads are an intermediate behaviour description technique that lie between Jacobson's use cases and other more detailed notations like Jacobson's interaction diagrams and Booch's timing diagrams. A timethread is more concrete than a use case because it considers the role of components internal to a system; a timethread is more abstract than control-flow because it can represent the pure sequencing of activities.

2.6.1 Critique: Aggregate Behaviour Models

Control-flow representations, like Booch's timing diagrams, presuppose that the responsibilities for managing the flow-of-control have been allocated to components. This may not

be the appropriate representation when one is interested in the sequence of activities that must be performed and not the explicit control over that sequencing, e.g., as during the earlier stages of design before the details of object collaborations have been decided. Control-flow is a low-level concept that focuses on the way a resulting system executes rather than on the patterns of behaviour through it; the resulting details may make it difficult to understand the typical behaviour patterns through a system. Message sequence diagrams, like Jacobson's interaction diagrams, are syntactically different but not semantically different from Booch's timing diagrams. Syntactically, interaction diagrams do not show explicit returns of control but the returns of control are implied in the sequence of messages. Therefore, interaction diagrams like timing diagrams, presuppose that the responsibility for the flow of control has been allocated to clients and servers.

Use cases are a high-level specification technique for typical interaction scenarios, but focus on the interaction with a system as a black-box and do not address the interaction patterns between the internal structures of the black-box. Timethreads are a graphical representation of behaviour that is suitable for describing typical interaction patterns through a system; however, they require a complimentary architectural specification if they are to be applied to collaboration groups.

2.7 Summary

An architectural view of object-oriented systems is concerned with the structure and behaviour of a system in operation; objects, object references, and methods are the basic units. The basic architectural view presented here is called the object architecture. A constructive view of an object-oriented system is concerned with how it is built using classes, inheritance, and polymorphism.

Object-oriented frameworks offer large-scale reuse because they embody interaction patterns between objects and collaboration groups which are implicitly reused when subclassing off of a framework.

Most object-oriented modelling techniques use a combination of E-R diagrams and communicating state machines. The E-R models are relatively static and do not easily ex-

press behaviour patterns and dynamic structure. State machine models are behaviour rich, but tend to bury larger behaviour patterns in the fragments of possibly many state machines.

Existing techniques for modelling collaboration groups fail to address how a complete system may be composed of groups, how objects participate in multiple groups, and how the end-to-end behaviour of a system may be expressed in terms of the collaboration groups. The ObjecTime tool is a notable exception, but much of the semantics of an Objec-Time model is buried in code fragments and state machines associated with objects.

Chapter 3: Architectural Concepts and Visual Notations

This chapter introduces the architectural concepts and visual notations on which this thesis is based. A meta-model for the architecture of software systems is presented. The meta-model describes the rules and relationships that govern a set of basic architectural concepts. Like a framework, the meta-model is an abstract description that may be specialized through extension. A visual notation for the meta-model is presented. The meta-model and visual notation developed in this chapter are extended and specialized in the next chapter to include the concepts used for role-based modelling. In this way, the meta-model acts as a foundation for the concepts and notations used in this thesis.

3.1 System Models

There are many ways of viewing a system and there are many models to support these views [15]. A system view is a narrowly focused projection of some aspect of a system's total design that emphasizes certain properties and either omits or abstracts from other properties. This is important because: (1) individual views are simpler than the design of the system as a whole, and (2) uniform models can be developed to support the views. Some examples of views that are important in software development include: an *architectural view* which emphasizes a system's structure and behaviour as it executes, an *end-to-end behaviour view* which emphasizes the sequence of activities and component activations across the internals of a system that ultimately generate a system response in reaction to a driving stimulus, a *black-box behaviour view* which is concerned only with the behaviour of a systems as seen at its edges, and a *functional view* which emphasizes the transfor-

mations performed by the system to produce output data from input data.

A model has a notation with semantic properties and is a means of capturing the elements of a view. Figure 13 illustrates that there are many different models available to support the many views of a system. The triangular notation represents an is-a relationship between models, e.g., an architectural model is-a kind of system design model. The figure illustrates only a few of the possible models and provides some concrete examples of models in some the categories, e.g.: MachineCharts [11] is an example of an architectural model; Timethreads [10] is an example of an end-to-end behaviour description technique; the extensional specification style of LOTOS [50] produces a black-box behaviour model because it defines the behaviour of a system entirely in terms of the interactions between the system and its environment; and data flow models [51] are primarily concerned with functional issues.

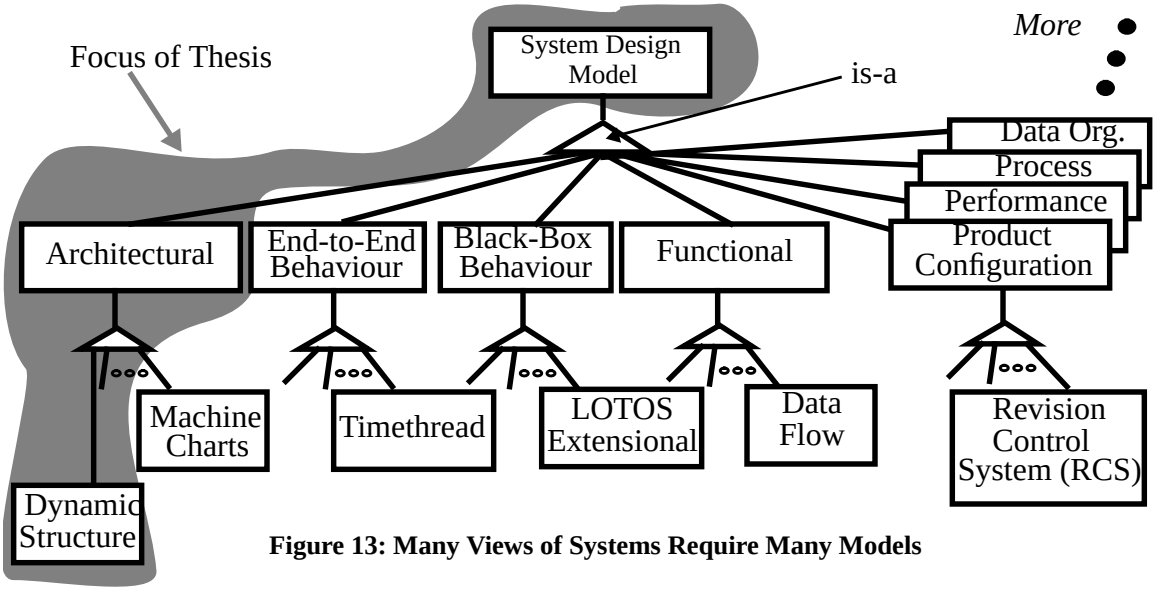

Figure 13: Many Views of Systems Require Many Models

Figure 13 divides the models based on the primary system view that the model supports. The result appears to be a collection of cleanly separated models; however, system views are not orthogonal and neither are the models that support the views. For example, an architectural view often has common elements to a functional view because the architecture of system must provide the infrastructure to support functional operations on data. As another example, a performance view may concern the performance implications of an-

other view, such as an architectural view or an end-to-end behaviour view. The result is that system models often have elements in common but the common elements are either expressed differently or are dealt with at a different level of detail in the different models. For example, the MachineCharts model has buttons that model functional operations, but the operations themselves are treated as stubs that take up time; the details of the operation are developed later or are expressed in another model.

The complexity of large software systems requires that developers use many different types of models throughout the production lifetime of a system [15]. Besides many different types of models, developers also use many different instances of the same type of model to address different parts of the system. For example, one may develop data flow diagrams for each of the major subsystems in a large system. The result is many overlapping model instances of the same and different types. One powerful development approach would be to aggregate the various model instances to form a complete specification for the design of the system from which an implementation of the system may be derived and maintained. This may be difficult because it requires that the models or aspects of them be formally defined and related. Also, formal notations, although powerful and unambiguous, are often difficult to use individually, because of their demand for completeness of detail [1], and together, because there may be conflicts between the details of different formalisms.

Another approach, and the one advocated here, is the use of multiple models, that may overlap but are consistent relative to one another, that may be informal, and that specify key system aspects and not the details required for a complete system specification. The use of multiple models and their integration is not pursued further; the thesis develops one model, a notation for it, and provides examples of its use.

The shaded region of Figure 13 highlights the type of model that is the focus of this thesis: an architectural model for the dynamic structure of software systems.

The *architecture* of a software system is a model of its operation in terms of its components and their communication pathways. There are two related aspects to system architecture: *structure* (components and their communication pathways) and *behaviour* (operation of and communication between the components). *Structure* is the decomposition of a sys-

tem into its components, the composition relationships among the components, and the communication pathways among the parts to achieve interaction. Example architectural components include subsystems, layers, objects, and processes. Structure alone does not include system operation (an element of behaviour) and is therefore a different concept from architecture. *Behaviour* is the temporal sequencing of the activities performed by a system and the communication patterns among the component parts and across the communication paths. Architecture does not explicitly address either *functional* concerns, such as algorithmic operations to transform data, or *data structure* concerns, such as the in-memory layout of data elements for efficient access; however, there are links to function and data in an architectural model. Functional operations on data appear in an architectural model in stub form: they are part of the architecture and are activated as needed, but the details of their algorithms are deferred. The major data elements in a system are considered part of the system's architecture which requires an *operational-view* of data, meaning that the major data elements are defined in terms of the operations that can be performed against them, and that the structure of the data elements is hidden by these operations.

Dynamic structure refers to changes that occur to a system's structure while it is running: components may be created and destroyed; the composition relationships between the components may change through time; and the communication paths among components may be created, destroyed or rerouted. A dynamic structure model allows these aspects of a system to be specified prior to system operation. Many software systems have limited structural dynamics, e.g., simple filters that transform data representations; or have fixed structures at a coarse-grained level, e.g., many large software systems are composed of a static collection of large subsystems. There are a class of software systems, however, such as those organized around object-oriented frameworks, that tend to have highly dynamic structures [13]. For such systems, modelling the structural dynamics may be important to understanding the system as a whole.

This thesis is concerned with collaborating groups of objects because of the importance of such groups in object-oriented frameworks. Collaborating object groups are architectural components, but are often only design concepts that are implemented in a distributed fashion in the code of several classes. The *structure* of collaboration groups includes: (1)

the composition of individual groups, i.e., the identification of a group's participants and any subgroups; and (2) the composition relationships among many groups, i.e., the communication pathways among groups and how participants are shared across groups. The *behaviour* of collaboration groups includes: (1) the causal-flow patterns among the participants of an individual group necessary for the group to fulfil its objectives, and (2) the causal-flow patterns across many groups necessary for the system to fulfil its end-to-end objectives. *Dynamic structure* of a collaboration group includes when the participants of a group play their roles relative to the other participants and to the group as a whole. Dynamic structure also includes the creation and destruction of collaboration groups and their participants, and the *movement* of participants among the collaboration groups.

3.2 Description of the Architectural Meta-Model

This section presents an architectural meta-model. A meta-model is a model that describes the properties of another model(s). The architectural meta-model describes the properties of models for specifying the architecture of software systems with emphasis given to their dynamic structure. Figure 14 presents the complete meta-model in Rumbaugh's E-R diagramming notation. Readers unfamiliar with this notation can refer to Appendix B. The concepts of the meta-model are the entities of Figure 14 and the relationships between the entities help to explain how the concepts interact.

The meta-model takes a component-based view of software systems in operation and uses a small set of basic concepts to support this view: places, terminals, wires, connectors, and messages. The meta-model describes the properties of these concepts, the relationships between them, and the constraints that govern their relationships. The concepts are sufficiently general so that they are not specific to a given model, but also extensible so that they may be customized for different models. The meta-model is extended by specializing the basic concepts in a manner analogous to subclassing in inheritance hierarchies: the specialized concepts inherit the properties and the relationships of the basic concepts.

A summary of the complete visual design notation that supports the meta-model concepts is given in Figure 15.The design notation uses visual shapes and annotations to distinguish the concepts of the meta-model, and uses spatial layout of the shapes and

connections between them to describe the relationships between meta-model concepts.

The sections that follow explain pieces of the architectural meta-model and develop visual design notations to support the meta-model pieces.

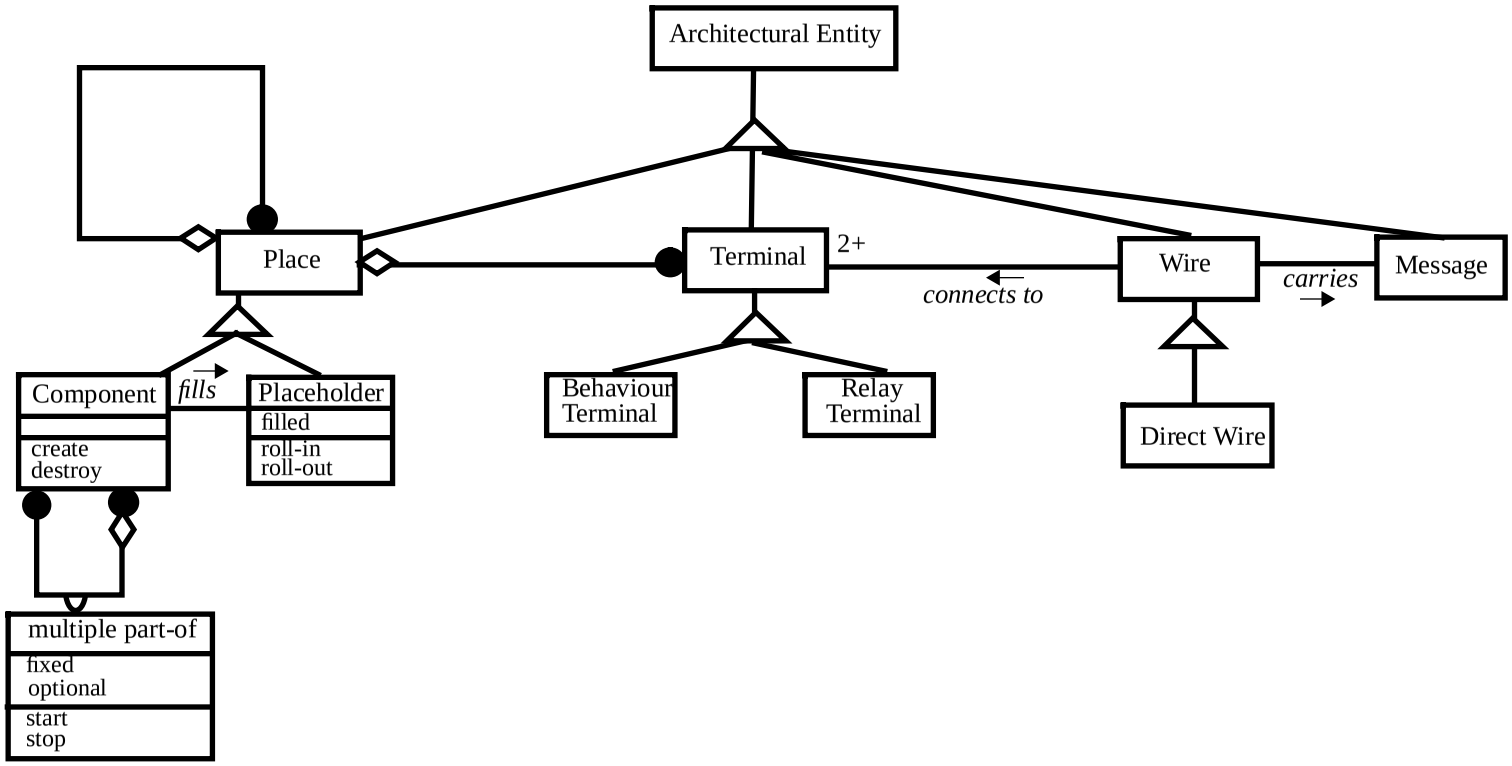

Figure 14: Meta-Model of Architectural Concepts

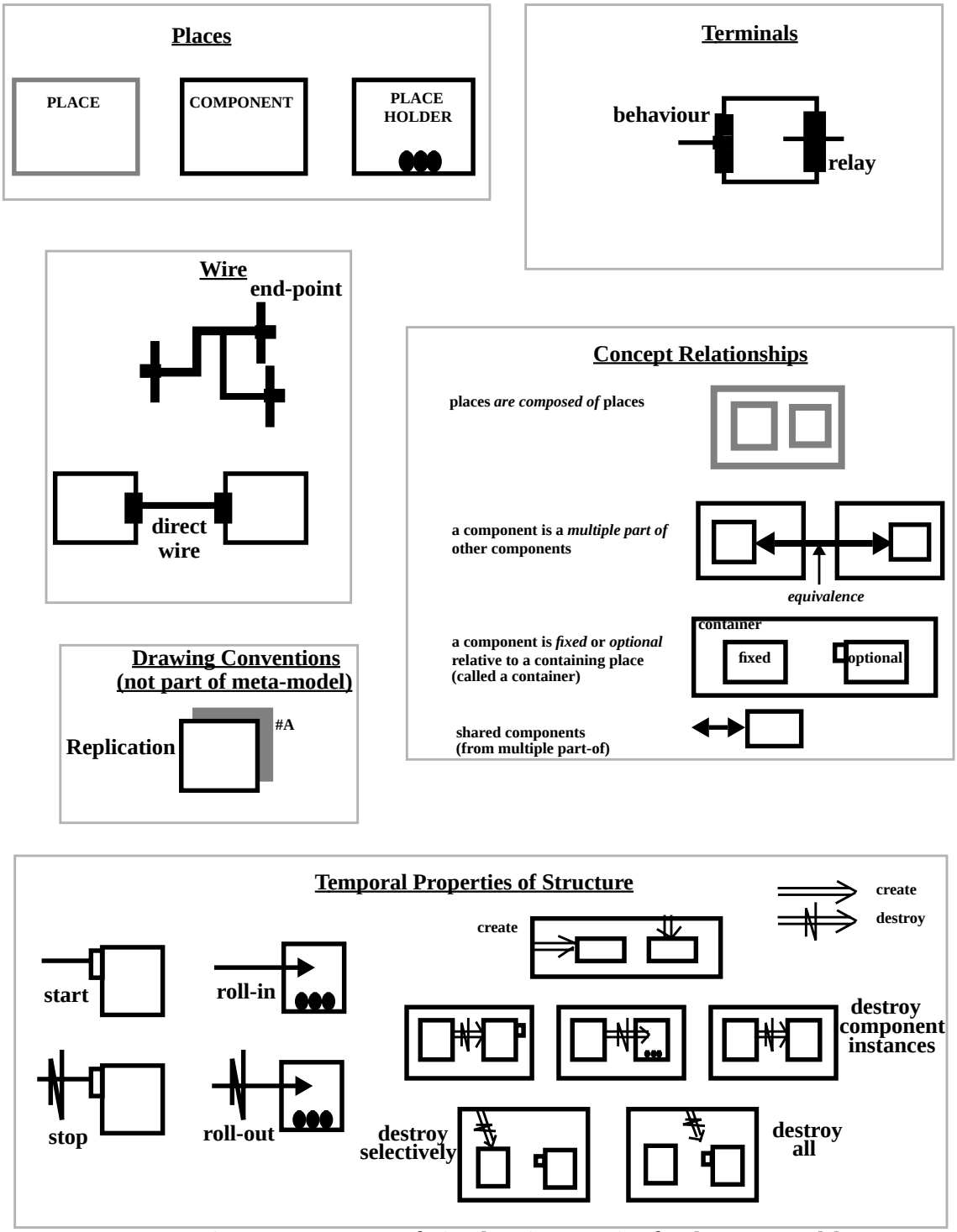

Figure 15: Summary of Visual Design Notation for the Meta-Model

3.3 Architectural Fundamentals

Places, terminals, wires, and messages are the basic architectural concepts of the meta-model. Figure 16 presents a portion of the meta-model, shown as shaded in the figure, that allows for the creation of simple design models in terms of the basic architectural concepts. The visual design notation for each of the basic concepts and a simple example that combines them are shown in the bottom half of the figure. The example illustrates a *wiring diagram* because it shows a collection of places that are joined at their interface points by wires.

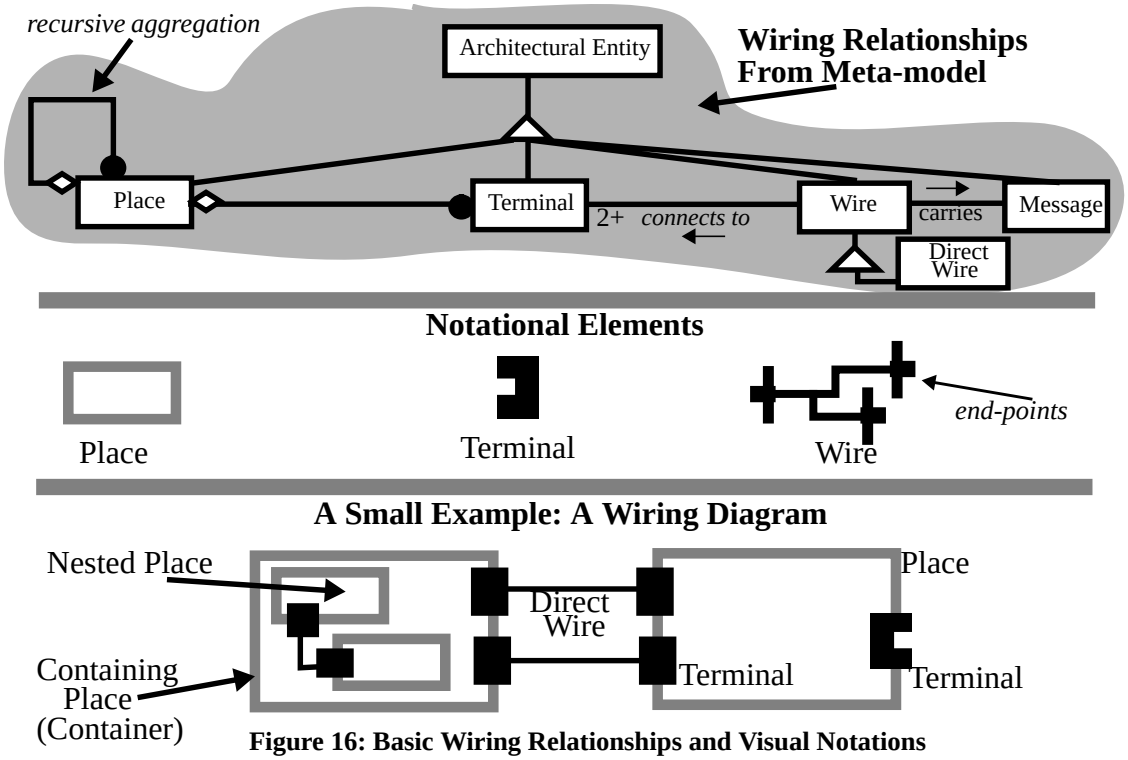

Figure 16: Basic Wiring Relationships and Visual Notations

An ***architectural entity*** is the most general concept of the meta-model. It provides only an *identity attribute* that is used to uniquely identify instances of architectural entities in a model of a system.

A ***place*** is a node in a wiring diagram; it is drawn as a box with a hashed outline. Ultimately places are where activities are performed; this distinguishes a place from a terminal

and wire which are concepts that do not perform activities. The recursive aggregation relationship on the place concept of the meta-model (see Figure 14) specifies that a place may contain other places. This supports the recursive decomposition of a system into successively smaller pieces, e.g., layers into subsystems and subsystems into objects. The decomposition of one place in another is represented visually by nested boxes. A place that has places inside of it is called a *container*, and a place inside of an another place is called a *nested* place. Places may have many terminals and the terminals may be connected by wires to the terminals of other places. A place is said to be a black-box because terminals on the edges of a place create an interface through which all information flows into or out-of the place must pass, and message passing through terminals is the only way that places (or more specialized kinds of places) may exchange information.

The place concept by itself does not have behaviour meaning that it is incapable of responding to requests made of it and may not spontaneously generate requests to other architectural entities. Therefore, the place concept is an insufficient abstraction for creating architectural designs. It is analogous to an abstract class in object-oriented programming, because instances of a place do not occur in design diagrams and because the place concept acts as a template for more specialized concepts.

A ***message*** is simply a container of information that flows over wires; one must look into the message to discover the meaning of the information. The contents of a message may include the identifiers of architectural elements and the values of primitive data elements. Deferred are issues of message copying, i.e., whether the contents of a message are copies or references to the information they represent. This keeps the message concept independent of, and open to refinement, implementation concerns like shared versus non-shared memory in distributed systems. A message is not given a notation, instead, the contents of a message are drawn as small flows shown next to a wire. The shape of the flow depends on the architectural entity it represents. Examples are given later.

A ***wire*** is a connection between components over which messages flow; it drawn as an arc with connectors on its ends. A wire has end-points that may be connected to terminals. A message originating at one of the end-points of a wire will travel to all the other end-

points on the wire. A good analogy is an ethernet cable with many taps into it. The general concept of a wire is undirectional; it supports message flow in any direction over it, however, particular instances of a wire may restrict the direction of message flow because of the properties of the terminals to which they are attached. A wire may not be decomposed into sub-wirings.

A ***direct wire*** is a refinement of the wire concept that is a point-to-point bidirectional communication pathway. It has exactly two end-points.

Terminals act as the interfaces of components and may be connected to wires. An attribute of a terminal is its *wiring protocol* which defines the legal messages that may be sent from it and received by it, and the rules that govern the sequencing of the messages. The wiring protocol of a terminal places restrictions on what terminals may be bound together by common wire. For example, a specific terminal expecting messages of type **X** may be bound to a terminal that sends messages of type **X**, but not to a terminal that sends messages of type **Y**. The exact form of the restrictions and how they are enforced are deferred to more specialized architectural concepts.

Although terminals have wiring protocols, they have no behaviour of their own; therefore, a terminal may neither perform activities nor interpret the meaning of the messages which pass through it. This separates the terminal concept from the place concept and its derivatives.

Figure 17 presents, from the meta-model, the refinements to the terminal concept and their visual design notations. A ***behaviour terminal*** is a terminal that exists on the interface of a place. It has only one side for connecting to wires: the side facing outwards from the place to which it is attached. A behaviour terminal is implicitly wired to the behaviour of the place to which it is attached (the term “place” is used here to refer to all possible specializations of the place concept that provide behaviour since the general place concept is without behaviour). A ***relay terminal*** also exists on the interface of a place, but wires may be connected to both of its sides. A relay terminal is used to wire an internally nested place across the interface of its containing place. The messages which pass through a relay terminal are not seen in any way by the place to which the relay terminal is attached.

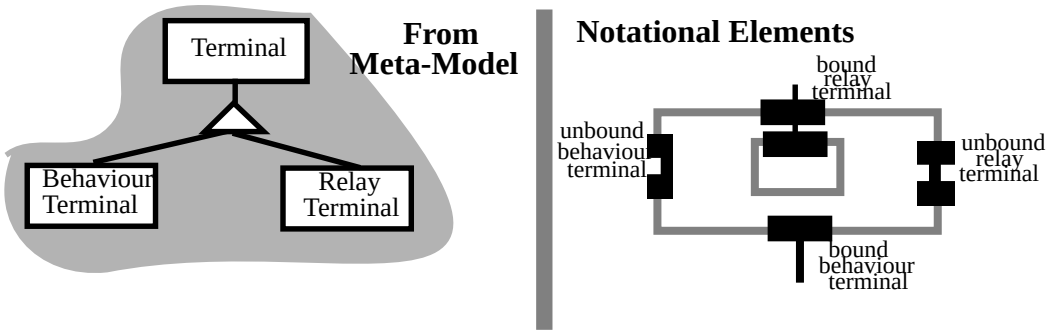

Figure 17: Terminals

Figure 18 combines the notations discussed thus far. The example of Figure 18 looks very much like a hardware wiring diagram. The boxes could represent chips, the terminals could represent pins, and the wires the signalling paths between the chips. Everything in this diagram is fixed in place. Figure 18, illustrates properties that are common to many software design models: black-boxes with interfaces, nesting of boxes, and connections among boxes to support interactions and the exchange of information.

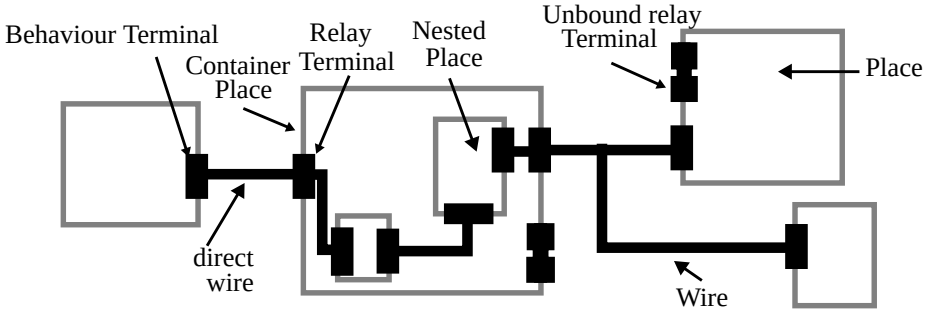

Figure 18: A Wiring Diagram with all Properties Fixed

3.4 Dynamic Structure: Components and Placeholders

Figure 19 introduces two new meta-model concepts, components and placeholders, that introduce behaviour and the basic concepts needed for modelling dynamic structure.

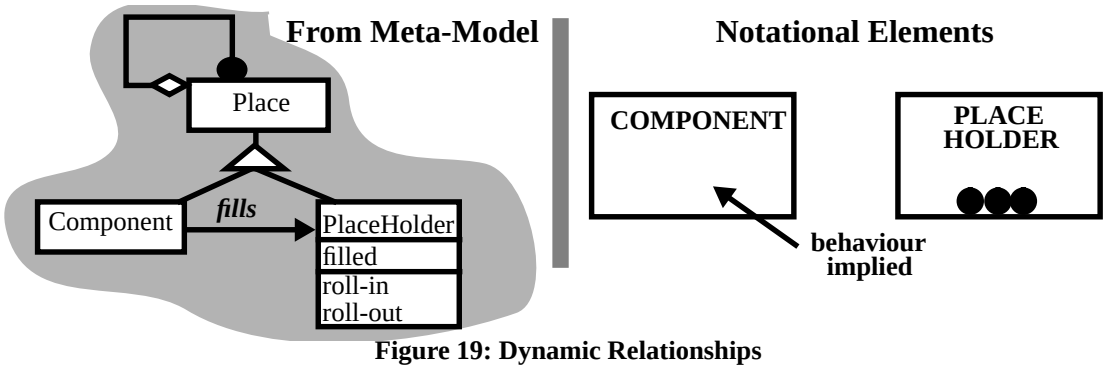

Figure 19: Dynamic Relationships

A ***component*** is a kind of place that may have its own behaviour. A component is drawn as a box with a solid outline to emphasize that it is a more concrete concept than a place. The edge of the box is used in design diagrams as the source of actions taken by a component's behaviour.

Components have structural properties inherited from the place abstraction, so components are black-boxes, have interfaces, and may be recursively nested. A component models a physical or conceptual entity that forms part of a designer's mental model of a system in operation [13]. Objects, processes, and subsystems are examples of components, but some are closer to the implementation than others.

All behaviour originates in components and occurs at places of the system that are occupied by components. The *operational lifetime* of a component instance begins in a place when it has acquired its necessary system resources and has initialized its internal state information such that it is capable of performing actions in the place either autonomously or when requested to do so, depending on its nature. There is an implied initialization phase for any component instance in a place, but the meta-model does not make the initialization phase explicit because it is concerned with the operation of the system in steady-state and is less concerned with the transients due to initialization operations. The initialization phases in implementations of architectural models may be very complex, because it is during these phases that the steady-state conditions of the model are achieved by lower-level concepts like pointer passing, memory allocation, and data assignment. There is also an implied termination phase associated with the end of a component instances operational

lifetime in a given place. A termination phase involves activities like freeing memory and invalidating pointer references in implementations of architectural models.

A ***placeholder*** is a reserved place in a wiring; instances of components fill placeholders during system operation. A component fills a placeholder by rolling into it using the *roll-in* operation of the placeholder concept in Figure 19, and the placeholder is said to be *filled* (i.e., its filled attribute becomes true). A component may be rolled-out of a placeholder using the *roll-out* operation of the placeholder concept in Figure 19, and the placeholder becomes *empty* (i.e., its filled attribute becomes false) and ready to accept another component. Conceptually, rolling a component into or out of a placeholder equates to the *movement* of a component and is an aspect of dynamic structure. A placeholder is drawn as a box with a set of rollers internal to it. The rollers symbolize that a placeholder may have components rolled into and out-of it during system operation.

A placeholder has no behaviour of its own, but it has structural properties inherited from the place abstraction. A placeholder has an interface of terminals that must be matched by component instances that are to fill the placeholder. This means that the interface terminals of the placeholder must be a compatible subset of the interface terminals of the component; i.e., a component must have a matching terminal for each terminal of the placeholder, but the component may have more terminals that are not used in the context of the placeholder. As with the compatibility of terminals bound by a wire, the exact rules governing the compatibility of a component's terminal to a placeholder's terminal are deferred to more specialized architectural concepts.

A placeholder has no internal structures that exist in it prior to being filled by a component; although components that fill placeholders may be nested to an arbitrary depth. Placeholders act as an interface specification for component instances that wish to participate in the context of the placeholder; the specification of the internals of the components is found in other diagrams. This property of a placeholder amounts to a constraint on the inherited recursive decomposition property of places as seen by placeholders.

When modelling systems, however, it may be useful to draw diagrams that show placeholders with internal structure. In these cases, the internal structures of the placeholder are

just design cues that help to explain the role of the placeholder in its context. For example, components may be rolled into a placeholder to have some data value they maintain updated. It may be more understandable to draw a representation for the updated data value in the placeholder. The specification of the placeholder informs the designer that components that fill the placeholder should provide something equivalent to the data value in their internal structure. The internal structure of a placeholder represents a typical organization of the internal structure of components that fill it and acts as a design cue or design constraint for other parts of the system. The operational semantics of the meta-model, however, do not consider the internal structure of a placeholder when rolling a component into a placeholder; a component and placeholder need only be interface compatible.

3.4.1 Placeholders: Roll-in and Roll-out

Figure 20 illustrates the notations used to represent rolling a component into and out-of a placeholder. Since a placeholder has no behaviour of its own, some component external to the placeholder must be responsible for rolling the component into it. In the example of Figure 20, the behaviour of the containing component, called a *container*, of the placeholder *rolls-in* a component instance into the placeholder at some point during the container's operation. This is shown by an arc drawn from the edge of the container to the internals of the placeholder, and a component instance is shown as flowing along the arc. The arrowhead on the arc is used to make explicit the control direction of the operation: from the tail of the arc to the head. Any component in the same container as the placeholder may perform the roll-in operation, but the black-box semantics of a place restrict components external to the container of a placeholder from doing the same. A roll-in operation causes a placeholder to be filled and causes the component filling the placeholder be begin operation in the local context of the container. A filled placeholder may have a component rolled-out of it. The *roll-out* operation is represented by an arc with a zigzag annotation across it; the arc points to the internals of the placeholder.

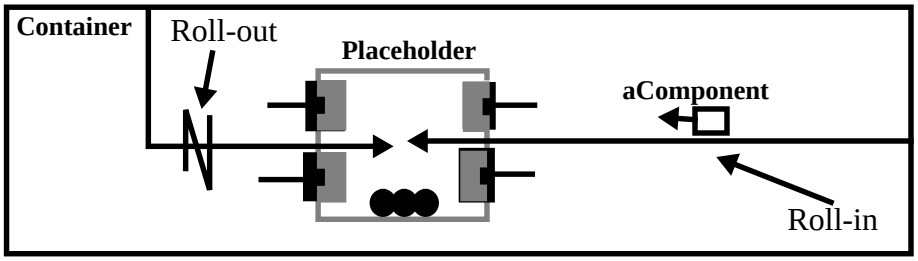

Figure 20: Rolling a Component into a Placeholder

Conceptually, rolling a component into or out of a placeholder equates to the *movement* of a component and is an aspect of dynamic structure. In Section 3.5 the architecture of a system is defined to allow a component to be in multiple places, perhaps simultaneously. Roll-in puts a component into a placeholder; it does not, by itself, cause the component to leave other places. This effect could be achieved by a roll-out operation in one place followed by a roll-in operation in another. Similarly roll-out does not imply that a component is moved to some other place: it simply means that the component filling the placeholder is no longer operational in the context of the container.

3.4.2 Fixed and Optional Components

A component may be characterized as *fixed* or *optional* **relative** to a containing component. This is specified in the meta-model by the **fixed** and **optional** attributes of the recursive aggregation relationship involving components in Figure 21. Modelling the fixed and optional properties of a component as attributes of the relationship between a component and a container is important, because it allows the temporal properties of a component to be expressed at a finer level of granularity then if the temporal properties were expressed relative to the system as a whole. The adjectives fixed and optional do not represent different types of components but merely serve to characterize a component in its relationship to one containing component. In the meta-model of Figure 14 the recursive aggregation relationship involving components has a black dot next to the container end of the relationship (omitted here). This extra annotation specifies that a component may be part-of **many** other components (multiple part-of). This section is concerned with the temporal properties of a component relative to one container; multiple part-of is explained more fully in Section 3.5.

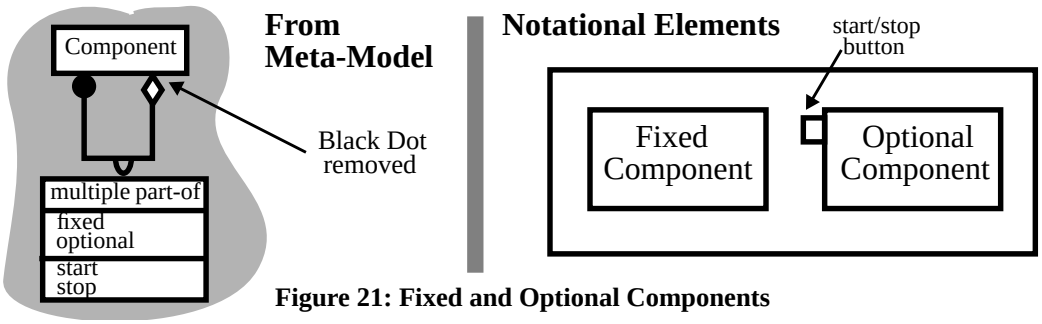

Figure 21: Fixed and Optional Components

A component is considered *fixed* if it is present and operational in a container when the container begins to operate and remains operational there for the operational lifetime of the container. A component is *optional* if it begins operation in a container sometime after the lifetime of the container begins. An optional component may have its operation stopped in a container before the lifetime of that container ends.

Figure 21 shows the notations used to differentiate these cases. A fixed component is represented as a box without a special annotation because fixed components are considered the default. An optional component has a small square box on its external edge that represents a *start/stop button*. In Figure 21, start and stop are modelled as operations on the multiple part-of relationship.To start a component means to begin its operational lifetime in a place, and to stop a component means to end its operational lifetime in a place. Fixed components are implicitly started and stopped at the end-points of their container's operational lifetime; however, these operations are applied explicitly to optional components.

An optional component is started by activating its start/stop button. This is represented by drawing a simple arc "pushing" the start/stop button; the source end of the arc is attached to the component that does the push, see Figure 22. An optional component may be stopped by "pushing" the start/stop button a second time with a stopping push (multiple successive starts have no effect). A stopping push is represented as an arc with a zigzag annotation through it. Once an optional component is stopped it may not be restarted; thus, a stopping push ends the operation of an optional component in a container. A start or stop operation on an optional component may only be performed by a component in the same container as the optional component.

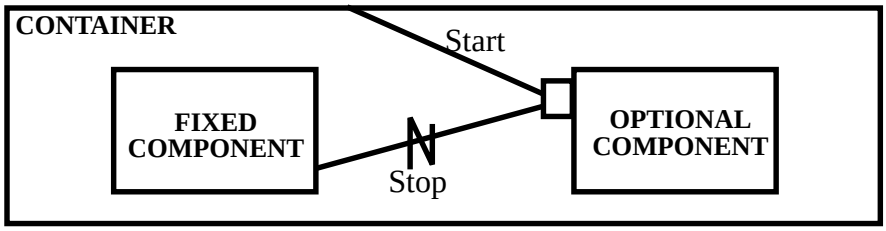

Figure 22: Starting and Stopping an Optional Component

Figure 23 illustrates the possible nesting relationships: an optional component in an optional component (**B** in **A**); a fixed component in an optional component (**E** in **A**); a fixed component in a fixed component (**F** in **E**); and an optional component in a fixed component (**G** in **E**). The temporal properties of an individual component relative to the system as a whole can be determined by considering the temporal properties of the component and the temporal properties of each component that contains it. Note that the fixed and optional attributes do not apply to placeholders because a placeholder is simply a prewired slot in a design diagram that can only be made operational by rolling a component instance into it. When a component is rolled into a placeholder it is implicitly started; when a component is rolled out-of a placeholder it is implicitly stopped in the context of the placeholder.

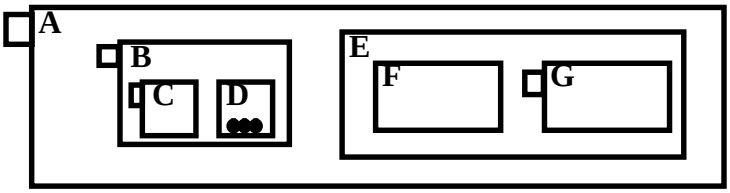

Figure 23: Dynamic Structural Properties may be Nested

3.4.3 Temporal Properties of Components in one Container

Figure 24 summaries the temporal properties that govern the operation of components and filled placeholders relative to one container. The figure shows a fixed component, an optional component, and a placeholder nested inside of a container. Also shown is a timeline that illustrates one possible scenario for the operational lifetime of the container. The thick vertical lines represent the starting and ending points of the operational lifetime of the container. The filled boxes along the time axes represent the lifetimes of component instances

that operate in the context of the container. The fixed component's operational lifetime in this context coincides with the operational lifetime of the container. The operational lifetime of the optional component in this context begins sometime after the lifetime of the container begins and ends sometime before the lifetime of the container ends. In the case of the placeholder, different component instances (shown by the different fill patterns in the figure) may roll into and out-of operation during the container's lifetime. The same component instance may fill a placeholder at different times (see the figure): the only restriction is that the same component may not be in two places simultaneously within the same container. When the lifetime of the container ends, any operational optional components and filled placeholders are implicitly stopped or rolled-out respectively.

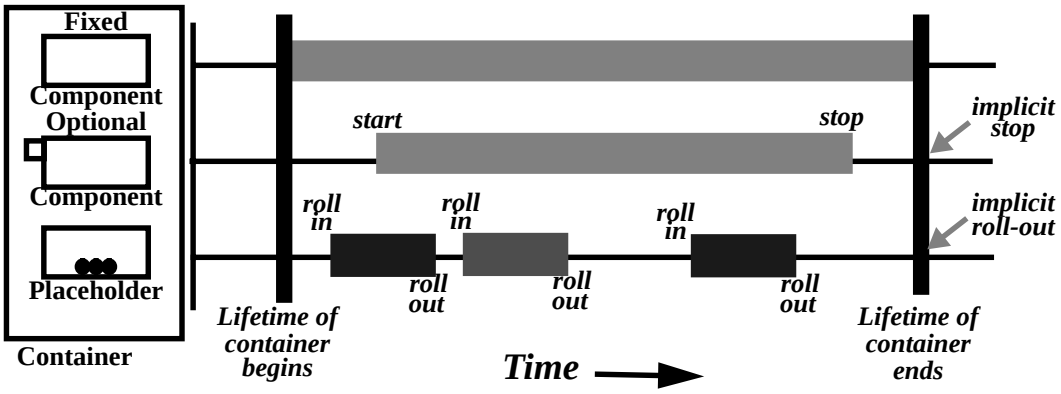

Figure 24: Temporal Properties of Components and Placeholders

3.5 Multiple Part-of: The Concept

This section describes the multiple part-of relationship among component entities of the meta-model.[4] The fragment of the meta-model dealing with multiple part-of is repeated in Figure 25. This is the same meta-model fragment that appeared in Figure 21 except for the black-dot that has been added next to the container end of the relationship arc. The multiple part-of relationship says several things: a component may contain many other components, a component may be contained in many other components (the addition of the black-dot allows for this), and a component may be fixed or optional relative to each of its containers. The notational elements in Figure 25 introduce a new notation to support multiple part-of,

4. The term *multiple part-of* comes from ObjecTime [45].

called an *equivalence arc*. An equivalence arc is a thick double-headed line that joins two or more places (components or placeholders); it specifies the places in a system that will contain the same component instances at runtime, though not necessarily simultaneously.

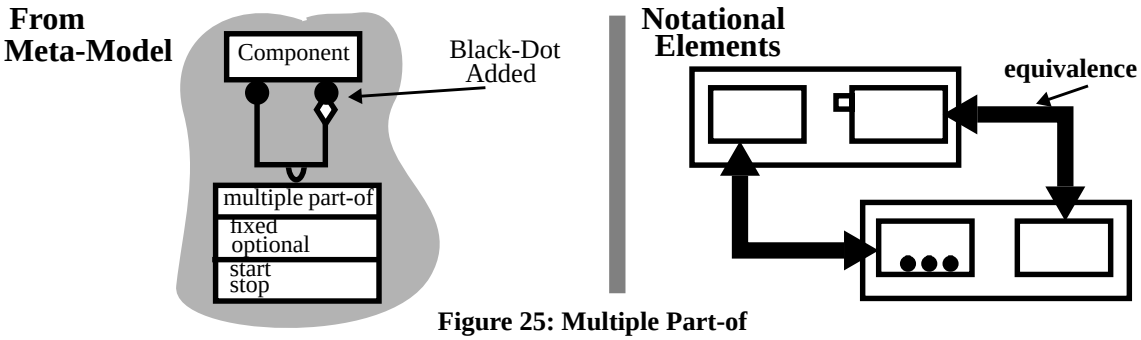

Figure 25: Multiple Part-of

Multiple part-of introduces two general cases: a component may be in different places at different times, or a component may be in different places at the same time. Figure 26 shows the simplest case where the same component (labelled **A** in the figure) is involved in two different decompositions at different times. Note that this requires the ability (conceptually if not physically) of a component moving from one place in the system to another. A good analogy is a distributed system with object instances that flow between the nodes.

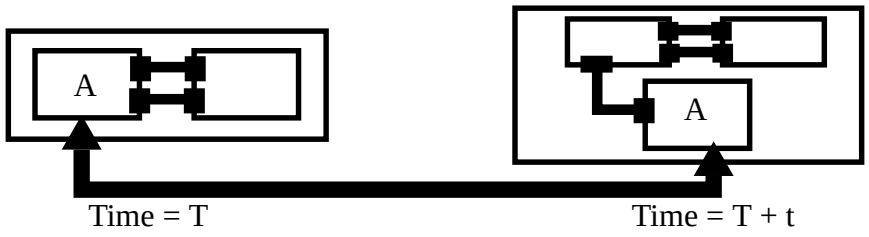

Figure 26: A Component may be in Different Places at Different Times

A component may also be involved in multiple decompositions at the same time, see Figure 27. This is useful in cases where a component has different roles that may be used to factor a system into a set of simpler views. For example, a figure object in a graphics editor may be involved in one factoring to draw an image of itself on the screen. In another factoring, the figure object may be participating with other objects to service user interactions.

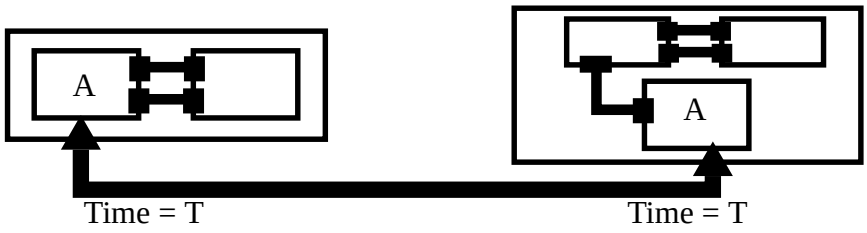

Figure 27: A Component may be in Different Places at the Same Time

One may imagine a factoring of components as a projection from a *wiring plane* to a *grouping plane*. Figure 28 shows one example. In the wiring plane, a component is shown in only one place, and all wires to the component are bound to this place. The grouping plane represents a factoring of the wiring plane into collaborating groups of components. The collaboration groups are represented as components. Boxes in the wiring plane and the grouping plane that have the same shading represent the same component. The grouping plane represents one possible configuration of the components in the wiring plane at a moment in time.

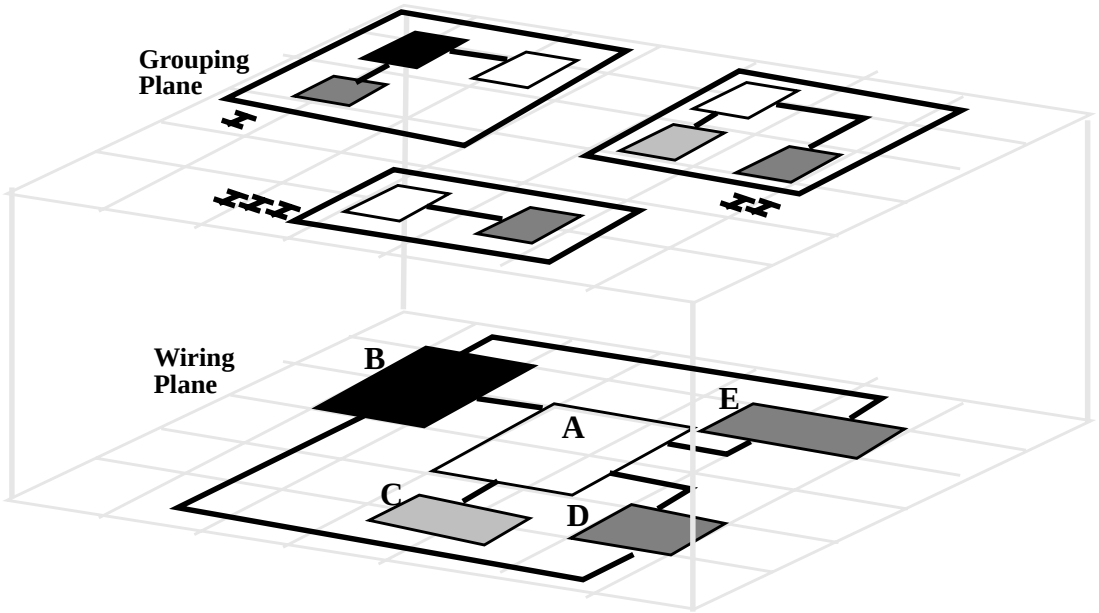

Figure 28: Factoring a Wiring Plane into Collaboration Groups

Factoring a system in this manner may seem to introduce unnecessary complexity. After all, there is only one instance of a component at runtime and it can have only one physical memory location (if we do not consider the distributed system case). Factoring a system in this manner has a number of advantages, however: (1) it simplifies an otherwise complex wiring into a collection of smaller and simpler units; (2) it enables a collaboration group to be treated as a component with an interface, identity, and possibly a behaviour of its own; and (3) it allows the temporal properties of a component to be specified relative to each collaboration group in which it is involved.

3.6 Multiple Part-of: Collaboration Groups

The grouping plane of Figure 28 represents one possible configuration of components at a point in time. It is possible, however, to specify the grouping plane in a generic way such that it acts as a template for a subset of the possible component organizations at runtime. For a large system, many such diagrams would be needed to address the different parts of the system. The purpose of this section is to show how the architectural concepts may be used in combination to create generic architectural models in terms of collaborating groups of components.

Figure 29 is a generic architectural diagram of the grouping plane shown in Figure 28. The collaboration groups labelled **I**, **II**, and **III** are the same in both figures. There is an equivalence relationship among the components labelled **A** in each of the collaboration groups. This specifies that the same instance of component **A** participates in groupings **I**, **II**, and **III**. **A** is always a fixed component as are the collaboration groups; therefore, **A** is operational in each of its places at the same time.

The remaining equivalence relationships in Figure 29 involve the placeholder **F** in collaboration group **I**. Since placeholders are initially empty, they must always be involved in at least one equivalence; the equivalence arcs specify the components that may be used to fill the placeholder at runtime. For example, the equivalence relationships involving placeholder **F** specify that either component **D** or component **E** may roll into it at runtime. The agency that performs the roll-in and the roll-out is the behaviour of collaboration group **I**. The equivalence relationships involving placeholder **F** specify that component **D** may be

part-of collaboration groups **I** and **II**, and that component **E** may be part-of collaboration groups **I** and **III**. Note that the multiple part-of relationship applies only to components (e.g., **D** and **E**) and not to placeholders (e.g., **F**) which are local to one container.

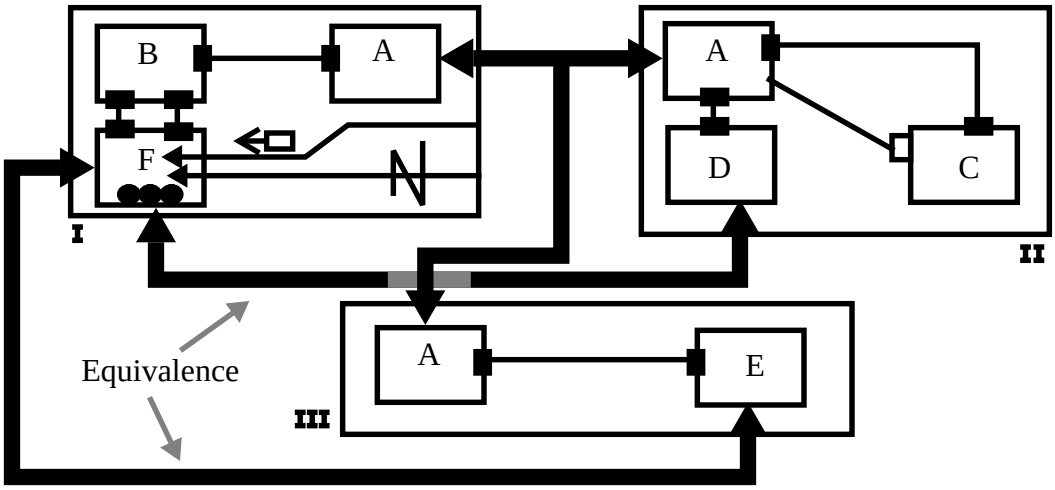

Figure 29: An Example Architectural Design Diagram

Figure 29 also specifies that component **B** is a fixed component that exists only in collaboration group **I**, and that component **C** is an optional component that is local to collaboration group **II** and started by component **A**.

3.7 Temporal Properties of Models

Architectural design diagrams, like Figure 29, act as templates for the runtime organizations that are possible. The templates embody both structural and temporal constraints. *Structural constraints* limit the possible runtime organizations in terms of component decomposition and component wiring. For example, in Figure 29, component **A** may exist in three places, **B** and **C** in one, and **D** and **E** in two. Furthermore, component **B** may only communicate with component **A**, **D**, and **E**. *Temporal constraints* place limits on the operational lifetimes of components relative to one another. For example, component **C** must begin to operate in collaboration group **II** sometime after **A** begins to operate in this same container because **A** starts **C**. As another example, components **D** and **E** may never participate in collaboration **I** at the same time. There are many behavioural characteristics that

are not specified by an architectural diagram like that of Figure 29: the sequence of activities performed by the components in response to stimuli; the interaction patterns between the components; and the causes and temporal orders of changes to the runtime configuration of components. Specifying this sort of behaviour requires another system view, e.g., an end-to-end behaviour view like timethreads.

3.7.1 Composite Lifetimes of Components

The semantics of components (both fixed and optional) and placeholders define temporal constraints on the operational lifetime of a component relative to one container. Through equivalence the temporal constraints on a component instance in all of its decompositions may be determined. The *composite lifetime* of a component is defined as the longest operational existence of a component if one considers all the decompositions in which the component is involved. At the beginning of the composite lifetime of a component there must be a *create* performed: to create a component means to acquire system resources and to give the component a unique identity. At the end of the composite lifetime of a component there must be a *destroy* performed: to destroy a component means to reclaim system resources and to mark the component's identifier as invalid. The composite lifetime of a component may have no gaps. This simply means that the same component instance may not be destroyed and subsequently created. Thus, a component's identifier must be unique for the lifetime of the system, regardless of the lifetime of the component, to avoid inconsistencies of interpretation. Create and destroy are operations supported by the component concept of meta-model (Figure 14); notations and additional semantics for create and destroy are developed in Section 3.7.2.

Figure 30 is used to describe the relationship between create/destroy and start/stop/roll-in/roll-out. On the left of the figure is an architectural design diagram. Components **A**, **B**, and **C** are containers of components **D**, **E**, and placeholder **F** respectively. **D**, **E**, and **F** are involved in an equivalence relationship, meaning that the same component instance will be "in" **D**, **E**, and **F** at runtime. The component instance is called **X** for convenience. On the right of the figure are two scenarios that represent possible temporal properties of the architectural design diagram. Each scenario is expressed as a timeline. Each axis of the timeline represents the composite lifetime of one of the components.The grey boxes on the

timeline represent the operational lifetime of the component **X** relative to the components **A**, **B**, and **C**.

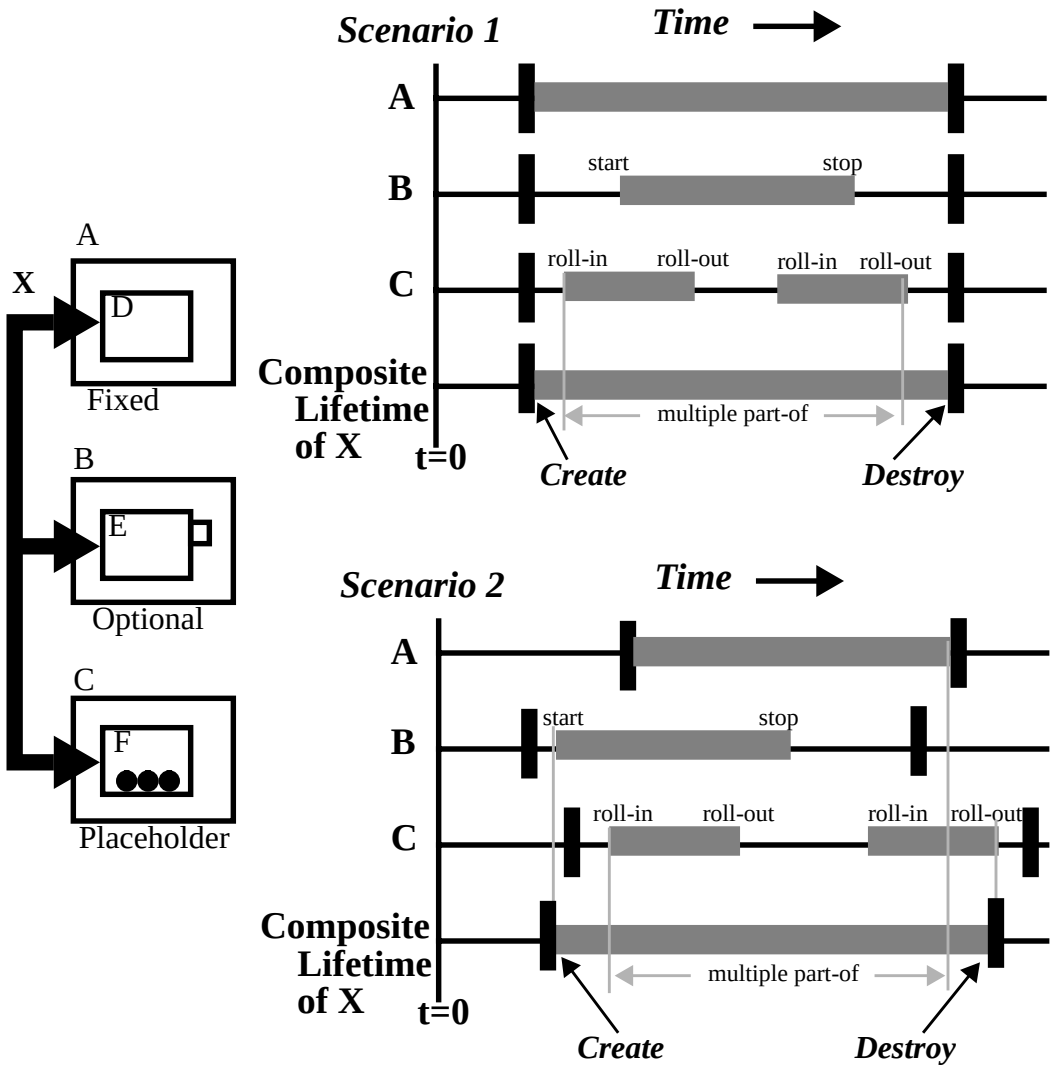

Figure 30: Composite Lifetime and Create and Destroy

In scenario 1, the composite lifetimes of all the components coincide. **X** is defined as fixed relative to **A** and is implicitly created when **A** begins operation. The creation of **X** is implicit because nothing in **A** needs to explicitly create **X**: the temporal semantics of components and equivalence, and the current structural state of the system provide enough information for the creation of **X** to be implied. When **A** ends, **X** is no longer operational in

any of its places and may be destroyed. The destruction of **X** is implicit with the end of **A**'s operational lifetime. From the time **X** rolls into **C** until it rolls out a second time, **X** is involved in multiple decompositions.

In scenario 2, the composite lifetimes of the components are not coincident. This could happen if the containing components **A**, **B**, and **C** were themselves involved in decompositions with dynamic properties. In this example, the composite lifetime of component **X** begins with the start operation in component **B**, because this is the first place that requires **X** to be operational. The roll-out operation in component **C** implies that **X** is destroyed as this is last place in which **X** is operational. (In scenario 1, the start in **B** did not imply a create since **X** was previously created when the composite lifetime of component **A** began.)

As general rules, a component is created when it is first required in any of its places, and a component is destroyed when its is no longer operational in any of its places. A create of a component may be coincident with a start operation of an optional component or coincident with the beginning of a container of a fixed component. A roll-in operation to a placeholder cannot imply a create operation because a component must be created in some other place before it may appear in a placeholder. A stop of an optional component, a roll-out from a placeholder, or the end of a container's lifetime may imply that a component is destroyed.

Figure 31 illustrates the temporal constraints on a component instance in every possible combination of places joined by an equivalence relationship. On the left of the figure are partial architectural design diagrams, and on the right of the figure are timelines that define some possible operational lifetimes of the components in the architectural design diagrams. **A** is a fixed component and **B** is an optional component started by **A**. The components or placeholders internal to **A** and **B** are joined in an equivalence. The component instance that will participate in **A** and **B** is called **X**. The axes of the timelines on the right of the figure illustrate the operational lifetimes of the component instances. The thick vertical bars represent the start-points and end-points of the lifetimes of components **A** and **B**, and the grey boxes represent the lifetime of **X** relative to **A** and **B**. *Case I, fixed equivalenced to fixed*: **X** is operational for the entire lifetimes of **A** and **B**. *Case II, fixed equivalenced to optional*: **X**

is operational for the entire lifetime of **A**, but **X** is started and stopped within the lifetime of **B**. The start of **X** in **B** must occur before the end of **A**, but the stop of **X** in **B** may occur after the end of **A**. *Case III, fixed equivalenced to placeholder*: **X** is operational in **A** for its entire lifetime, but **X** may roll-in and roll-out of **B** several times. **X** must roll into **B** before the end of **A**, but may roll out of **B** after the end of **A**. *Case IV, optional to placeholder*: **X** is operational within the lifetime of **A**, but **X** may roll-in and roll-out of **B** several times. **X** must roll into **B** before the stop of **X** in **A**, but may roll out of **B** after it is stopped in **A**. *Case V, optional to optional*: **X** is operational within the lifetimes of **A** and **B**. **X** must be started in **B** before it is stopped in **A**. *Case V, placeholder to placeholder*: **X** may roll-in and roll-out of **A** and **B** several times. The implication in this case is that at least one of the placeholders is involved in another equivalence to define a source for an actual component instance. There are no constraints on the ordering of the participation of **X** in **A** and **B** in this case.

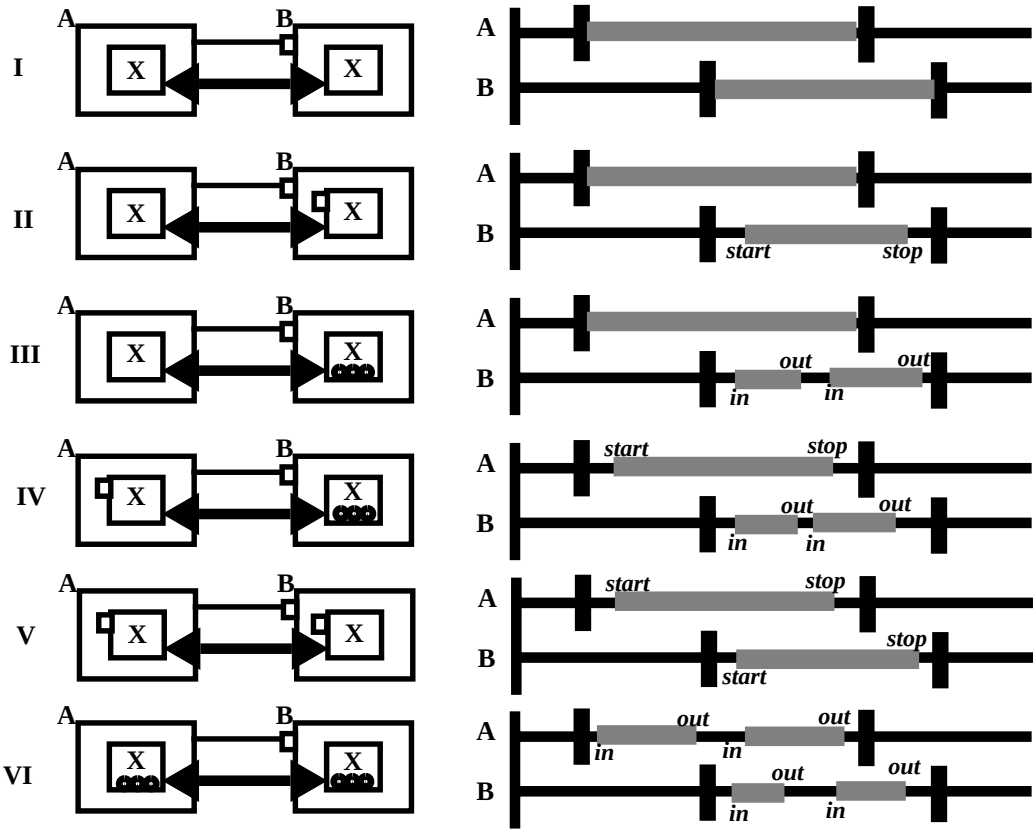

Figure 31: Equivalence and Temporal Constraints

The introduction of multiple part-of and equivalence means that components may be further characterized based on their participation in multiple decompositions. Figure 32 illustrates the possible cases. Components that are involved in only one decomposition are called *fixed-local components* or *optional-local components*. Components that are involved in equivalence relationships, and therefore in multiple decompositions, are termed *fixed-shared components* or *optional-shared components*. In Figure 32, the shared components have a small double arrow as an annotation. When a group of components is drawn in isolation, this annotation is helpful for indicating that a component is shared in a larger context. A placeholder must always be involved in at least one equivalence relationship to define which component instances fill it; therefore, no special annotation is needed to indicate the shared properties of a placeholder.

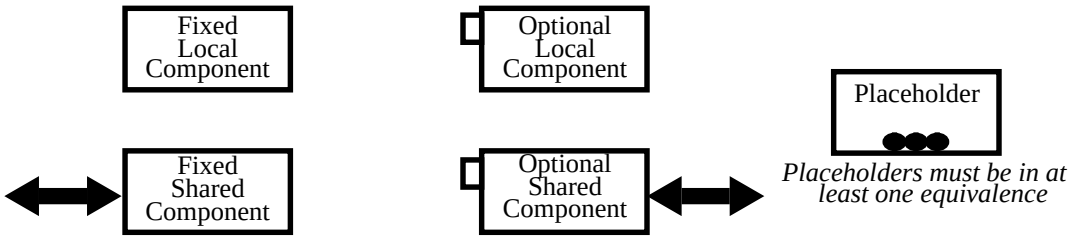

Figure 32: Characterizations of Components and Placeholder

3.7.2 Explicitly Creating and Destroying Components

Explicit creation and destruction of components are supported by the create and destroy operations of the component concept from the meta-model, see Figure 33. A create operation is drawn as a double lined arc with an arrowhead pointing at the component to be created. A destroy operation is similar but has a zigzag annotation across it.

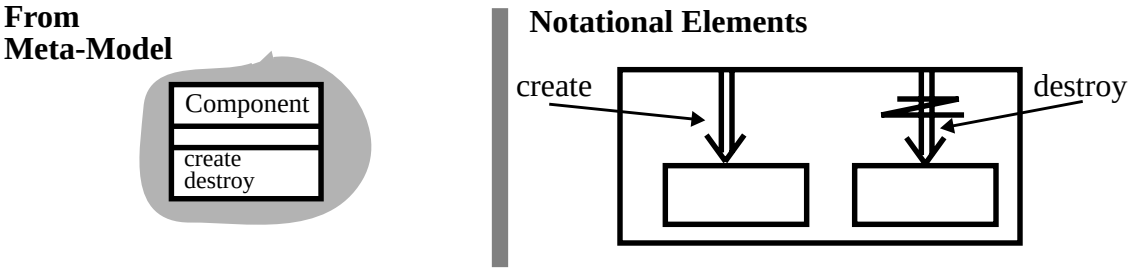

Figure 33: Explicitly Creating and Destroying Components

Create and destroy are simple low-level concepts that may be implied in an architectural design diagram as the example of Figure 30 illustrates. The extra information provided by equivalence relationships and the semantics given to places (components and placeholders) allows the start-point and end-point of a component's composite lifetime to inferred. This would allow a runtime system supporting these semantics to perform the necessary creates and destroys of components implicitly. This is analogous to languages with garbage collection, e.g., Smalltalk, that enable systems to be built without the explicit destruction of components. The architectural model proposed here enables this, but also enables systems to be built that are free of explicit creates.

There are cases in architectural designs, however, when it may be useful to show explicitly the creation or destruction of a component. Explicit component creation is useful in *initialization diagrams* that depict the transient system states needed in implementations to establish the context of a containing component in an architectural design. It is during these initialization phases that components are physically created and the identifiers of components exchanged in implementations. Although, such details are not the focus of the meta-model, initialization diagrams explain how an architectural model is physically implemented, which may help a designer understand the relationship between an architectural design and its implementation.

Figure 34 provides a very simple initialization diagram for a keyboard event handler component. The creation arcs from the edge of **KeyboardEventHandler** component specify that the **InterruptHandler** component and the **EventBuffer** component are created when the **KeyboardEventHandler** is created. The instance flow named **anEventBuffer** says that the **InterruptHandler** component is initialized with the identifier of the **EventBuffer**. This would allow the two components to communicate.

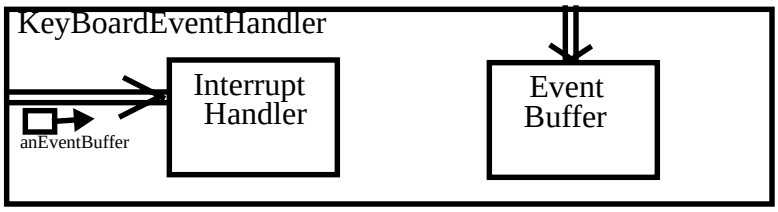

Figure 34: Creation in Initialization Diagrams

Explicit creation is also useful in cases where a container component acts as a factory for creating component instances that shipped off to other places in the system to do useful work. For example in a graphical editor application, a single component may be responsible for creating instances of drawings, drawings being components that maintain the figures to be displayed. Drawing components, once created, are shipped to other places where they are manipulated. In this case, making creation explicit clarifies the role of the factory component. The semantics of start/stop and roll-in/roll-out would be sufficient to specify this situation, but using them would obscure an obvious design intent.

Explicit destruction may be useful on diagrams that show the termination phase of a component's composite lifetime. Explicit destruction may also be useful in architectural designs when there is some master container of a component that defines the context which determines the end of a component's composite lifetime, even when a component may be involved in multiple decompositions. For example, a figure component in a drawing editor may be involved in many decompositions (e.g., to draw on the screen and to maintain invariants to other figures), but if there is a drawing component which contains all figures and this drawing component is destroyed, then the figure must be destroyed also.

Figure 35 illustrates the notations used for explicitly destroying components. Any agency internal to the same container as another component may destroy that component. It is possible to destroy an optional component and a component that is filling a placeholder. In the case of a placeholder, the destroy arc points to the internals of the placeholder to symbolize that the contents of the placeholder and not the placeholder itself are destroyed, see Figure 35. When an optional component is destroyed it is implicitly stopped; therefore, a destroy operation may not be followed by a start operation. When a component in a place-

holder is destroyed that component is implicitly rolled-out of the placeholder; the placeholder may be refilled at a later time with another component.

Destroying a fixed component is more complex than either of the other two cases. Destroying a fixed component causes the operation of that component to cease and *causes the destroy operation to be propagated to the container of the fixed component*. Defining a component as fixed relative to a container means that the component is required for the lifetime of the container; therefore, destroying a fixed component is like removing a vital organ of its container, and the container must die as a result. If the fixed component is replicated and the replication factor is not specified (i.e., the number of replicated components is optional), then all replicated instances must be destroyed before the destroy operation is propagated to the container of the replicated components.

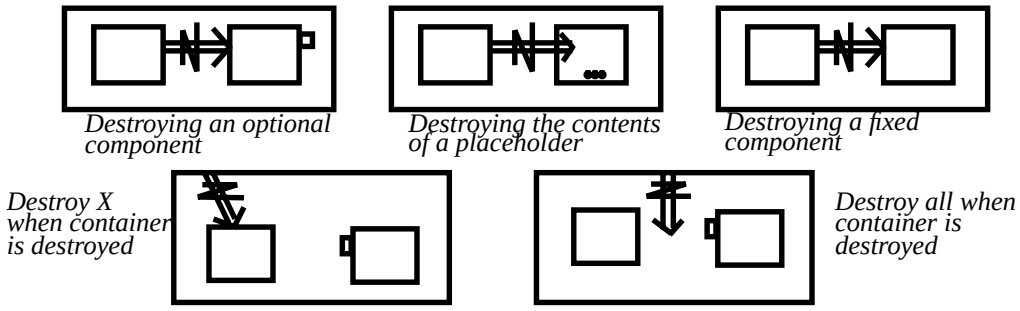

Figure 35: Notation for Explicitly Destroying Components

It is often useful to propagate a destroy operation to the internals of a container when the container itself is destroyed. The container may be destroyed explicitly as a result of a destroy operation applied to it or implicitly because it is no longer operational in a place. The bottom half of Figure 35 shows two different possibilities when a container is destroyed. The container may wish to selectively destroy individual components when it is destroyed, shown by an arc from the edge of the container to the to be destroyed component; or the container may destroy every internal component, shown by an arc from the edge of the container pointing to an empty spot in the container's internals. When there are no destroy arcs from the edge of a container, the default action is that none of the internal components are explicitly destroyed. The fact that the container is destroyed, however, may cause some of its internal components to be implicitly destroyed if they are no longer op-

erational in any place.

The notations for destroying components may be combined. In Figure 36, component **B** destroys fixed component **C**. This causes the destroy operation to be propagated to the container **A**. The destroy annotation of **A** specifies that all of its internal components are to be destroyed when **A** is destroyed. Thus the destroy operation is propagated to component **B**, **C**, and the contents of placeholder **D**. This introduces a circularity problem for the destruction of **C**, but such cycles are easily broken by maintaining the state of a destroy operation as it proceeds. Further destroys are possible, for example, the contents of **B** and **C** themselves may be destroyed if **B** and **C** are defined such that the destroy operation is to propagate to their internals.

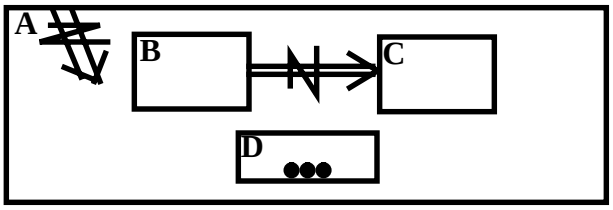

Figure 36: Combining Destroy Notations

When a shared component is destroyed, the destroy operation is propagated to all places in which the shared component currently exists. In the example of Figure 30, the shared component **X** is explicitly destroyed in container **B** where it is defined as optional. This causes the operation of **X** to cease in all of its places: **X** is stopped in **B** and **A**, and rolled out of **C**. The destroy operation is propagated to **A** when **X** is destroyed because component **D** is defined as fixed relative to **A**.

The meta-model does not define the semantics needed to destroy a currently active component that may be in communication with other components (the Ada programming language definition [54] provides an example of the type of semantics that are needed).

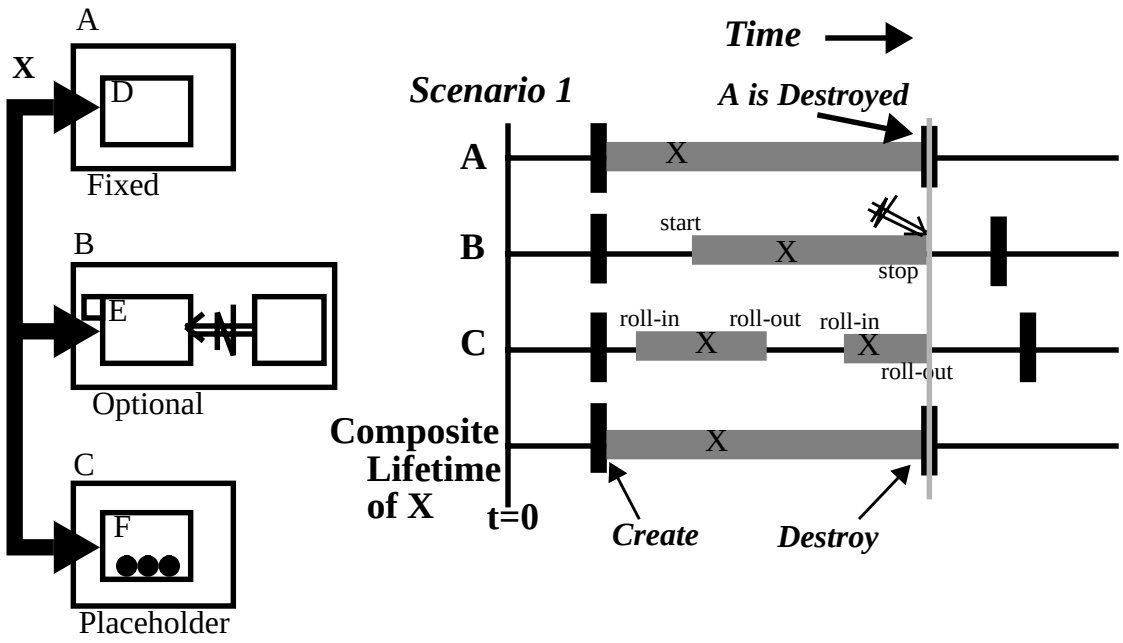

Figure 37: Temporal Semantics of an Explicit Destroy

3.8 Summary

The idea of a meta-model for the architectural concepts of software systems represents an important contribution of this thesis. The meta-model is useful because it: (1) helps to define terminology, (2) could be a basis for a general theory for the architecture of software systems, (3) could be a basis for an object-oriented framework for prototyping different models of software architecture, and (4) could be used to categorize and compare existing architectural models of software systems. The meta-model is inspired by research done in the DOORS project [13], the ObjecTime tool [45], Helm's contracts [26], and other research efforts, e.g., Hermes [47].

Chapter 4: Role-Based Model

This chapter describes the role-based architectural model and provides two small examples of its use. Timethreads are used to describe the causal flow patterns through the examples. The complete role-based model is described by extending the meta-model of Chapter 3. A complete summary of the notations developed for this thesis is given.

4.1 Object Architecture and the Role Architecture

Figure 38 (taken from [13]) illustrates the possible dynamic nature of an object architecture. In general, the system consists of an ever changing set of objects and object references over time. The object references are established through the exchange of object identifiers. When a new object is added new references are needed so that the new object may communicate with existing objects (e.g., when object **8** is added references are created to object 2 and from object **1**).

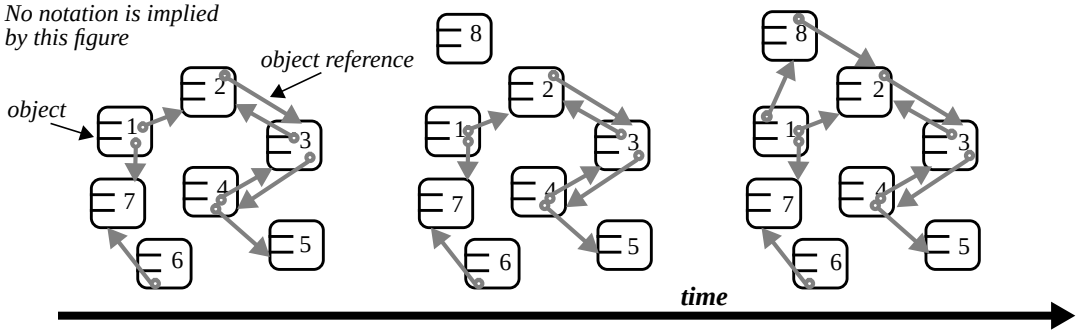

Figure 38: Dynamically Changing Object References Through Time

The object references form a basis for establishing, dynamically, collaboration groups

involving object instances. Figure 39 shows the nature of a system in terms of dynamically changing collaboration groups over time. In this view, the system consists of a set of objects that may come and go and that have dynamically changing groups overlaid on them. The left-side of the figure shows three such groups: **A**, **B**, and **C**. **A** could represent a Model/ View/Controller (see Section 4.5) relationship among components **1**, **2**, and **8**. Group **B** could be some parent/child relationship between objects **2** and **3**. The overlapping of the groups implies that object **2** participates in groups **A** and **B** simultaneously.

Figure 39 (taken from [13]) illustrates that the collaboration groups have temporal properties. The same group can exist at different times with different objects (e.g., group **B**). Objects might occupy a different role in different occurrences of a group (e.g., object **3** may occupy different roles in the instances of group **B**). Groups may be destroyed (e.g., group **A** and **C**) and created (e.g., group **D** and **E**).

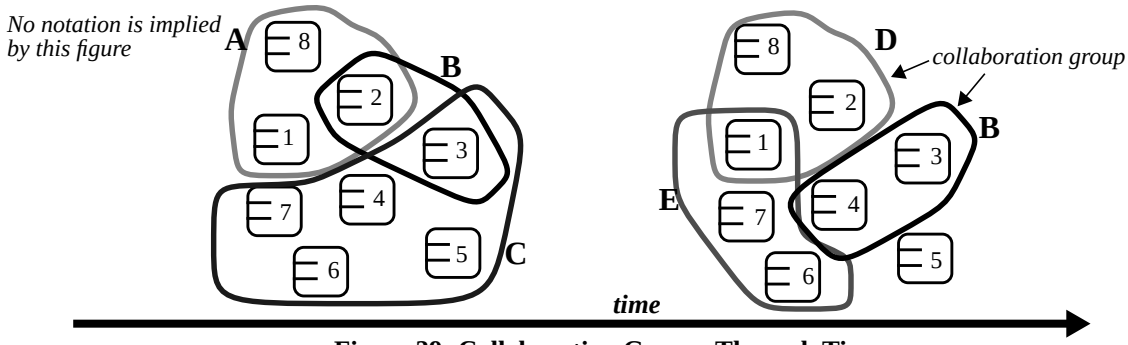

Figure 39: Collaboration Groups Through Time

Implicit in Figure 39 are the object references between object instances. Note that there may not be a one-to-one correspondence between an object reference and the group(s) it supports. For example, the same references among objects **1**, **8**, and **2** may be used to support both groups **A** and **D**. The result is that the object references may be relatively static compared to the groups they support. It may also be the case that two objects are involved in a group without an explicit reference between them. In this case the components must rely on alternate routes, perhaps through objects of the same group or through pathways external to the group in question.

Viewing object-oriented systems in terms of ever changing objects, object references,

and collaboration groups lacks generality, because there may be no apparent pattern in the coming and going of objects and collaboration groups. It may be necessary to project the system into a more general and fixed model, such as is offered by the role-based model.

Figure 40 relates the basic object architectural view with collaboration groups (shown on the left of the figure) to the role-based model (shown, conceptually, on the right of the figure). On the right of the figure, the thick outlined boxes represent *role teams* and the boxes with rounded corners represent *roles*. Activity in the role-based model emanates from objects that are playing roles in the teams. Role playing may have a dynamic nature, e.g., an object may play a role for a period of time and then leave that role to play another. There may be constraints on role playing, e.g., the same object may be required to play two roles simultaneously or play a given role for its entire lifetime. *Role wires* connect the interfaces of roles and allow the objects playing the roles to exchange messages.

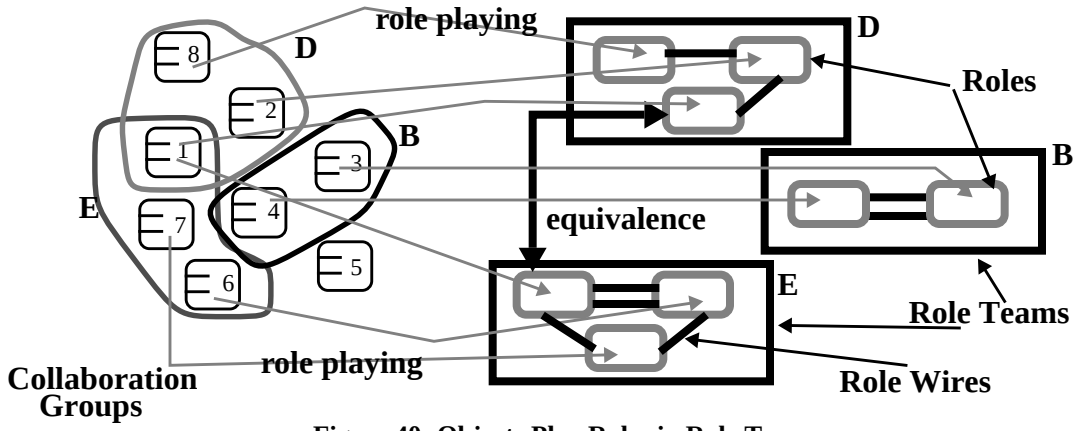

Figure 40: Objects Play Roles in Role Teams

A role-based model serves a dual purpose during system development: it can have either a definitional purpose or an operational purpose. A role-based model is *definitional* because it acts as template for many runtime organizations. Role teams are templates for collaboration groups and roles are templates for objects that play roles. Further, roles define the properties of objects that participate in them. The aggregate properties of an object may be determined by examining all the roles it plays. Roles may be seen as similar to classes, except that a role defines only those properties that are required of an object in the context of one role team. A role is different from a class because many objects from different class-

es may satisfy a role, and a single object may satisfy many roles. However, an object may belong to only one class. [1]Anderson [2] is using this relationship as a means of developing class hierarchies by aggregating the properties of shared roles (meaning roles played by the same object) into class definitions. A role-based model is *operational* when roles and role teams are considered to be components of a system's architecture. For example, when a scenario is played out, perhaps by tracing causal flow paths through the role team organization, the role teams are acting as instances of collaboration groups and the roles as instances of objects.

This duality of purpose is similar to the relationship between classes and objects in object-oriented design methods. For example, in Responsibility Driven Design [7] cards are used to document the responsibilities and collaborations among classes. The cards are called Class, Responsibility, and Collaboration (CRC) cards. A card is a class when it is sitting idle as a member of a set of cards. A concrete scenario is played by treating the cards as actors in a play. When it is a card's turn to perform its part (i.e., discharge a responsibility) it is picked and its responsibility is read (performed). At this point a class has become an object that is discharging a responsibility [6]. At the code level, the duality of purpose is less evident because classes are explicit but the roles that objects of those classes will play and the corresponding collaborations can be difficult to see.

This thesis is concerned with the operational aspect of a role architectures. We do not consider how classes may be constructed from roles nor how the definitional properties of an object (its interface signature) may be verified such that it may play a role.

4.2 Basic Elements of Role Architectures

Figure 41 illustrates the basic elements of the role-based model that are used in the example of Section 4.4 and Section 4.5. The top half Figure 41 shows the extension to the meta-model supporting the role-based model. Several new kinds of components are derived: role teams, roles, component-bound roles, and objects. Role wires are introduced as a refinement of the Direct Wire concept. Missing from Figure 41 are the elements of the meta-mod-

[1.] This is true of most inheritance-based object-oriented languages. Prototype-based languages are more flexible in this regard.

el and role architecture that are necessary for modelling dynamic structure. Section 4.7 re-introduces concepts like multiple part-of, equivalence, start/stop, create/destroy, and place-holders in the context of role-based modelling.

The bottom-half of Figure 41 shows the notational elements that correspond to the concepts in the top-half of the figure.

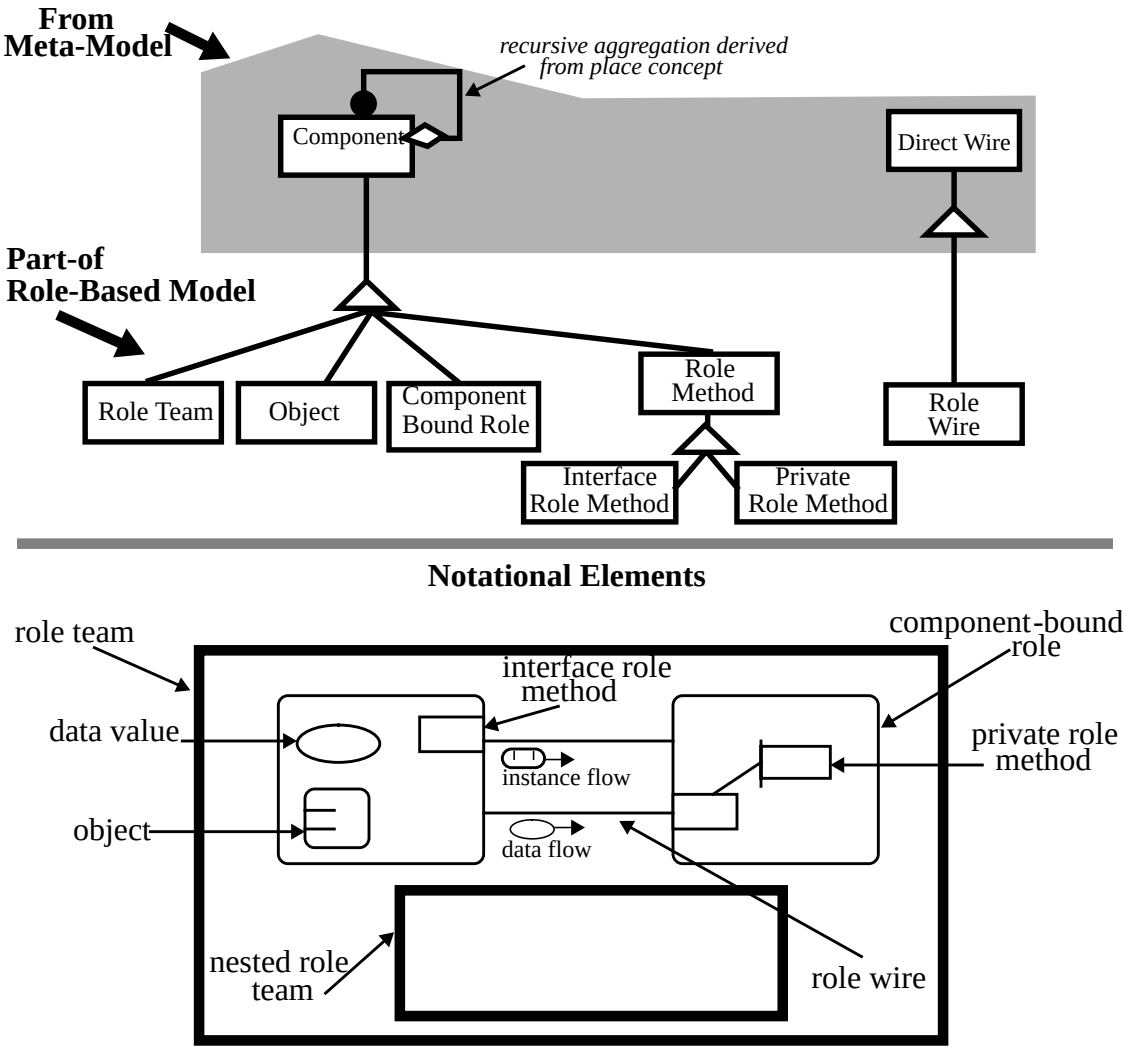

Figure 41: Elements of the Role-based Model

A *role team* is a component; it describes how a collaboration group is structured as a collection of interdependent *roles* and possibly nested role teams. The purpose of a role

team is to accomplish some objective. The objective may be to maintain some invariant, such as the connection of arcs to other figures in a graphics editor, or to perform some system-level behaviour, such as logging memory utilization reports in a real-time system with limited resources.

A *role* is a place in a role architecture; it defines the aspects of an object necessary for the object to participate in a role team. An object must adhere to structural and behavioural obligations to play a role. The structural obligations are defined by a role's interface, the wirings between roles (*role wires*), and the internal structure of a role. The behavioural obligations are the activities that an object playing a role must perform. Activities are similar to responsibilities in Responsibility driven design [7].

In Figure 41 roles are called *component-bound*. A *component-bound role* is one that is played by the same object for the operational lifetime of the role in its role team. For example, a keyboard manager which accepts events from a keyboard is an example of a role that may always be played by the same object. Recall that the operational lifetime of a component may be either fixed or optional. For the current discussion, we are considering only fixed component-bound roles. Section 4.7 introduces the concepts derived from the metamodel that allow for dynamic role-playing.

Role wires (Figure 40) are static connections between the interfaces of roles in a role team over which messages between objects playing roles flow. They are achieved in the implementation by object references but abstract away from details like the passing of identifiers.

Role methods are the destinations of messages sent across role wires. When a role method receives a message, it performs some activity based on the contents of the message. The activity may update local attributes, perform architectural operations on other components (Section 4.7), and send messages. A *private role method* is internal to a role and invisible outside of the role. The elongated edge of a private role method is for receiving messages and outgoing messages may originate from the remaining three sides.

The *attributes* of a role may be either data values or objects and appear internal to roles.

An *object* is drawn as a box with rounded corners with two lines on its interface. Objects represent entities that are part of the application domain of the system being modelled, e.g., the figures that are drawn on the screen in a drawing package. They appear explicitly in a role-based model when they are the attributes of a role or as instance flows, and implicitly by playing roles. As role attributes, objects may have their values read or updated, and they may be created and destroyed. Object attributes should not perform complex operations (system objectives) in a role model that require collaborations with other objects. This sort of behaviour should be expressed using roles and role teams. To do so requires that an object attribute also play a role in a role team (more on how to represent this in Section 4.7).

A *data value* represents either primitive data elements, e.g., integers and strings; or more complex data elements that are outside the application domain of the system being modelled, e.g., the use of stacks and queues to hold figures in a drawing package. In both cases, the abstraction provided by the data value is well known and the details of how the abstraction is supported through interface methods is unimportant for understanding the design of the system at hand. For these reasons, a data value is drawn without interface terminals; it is drawn as an oval and given a label to reflect the abstraction it supports. However, a data value may be implemented as an object in programming language terms. Data values never play roles.

Role attributes may flow over wires when they are the contents of a message. An *instance flow* represents an object identifier passed in a message and it is drawn as a small object icon with an arrow to indicate the flow direction. A *data flow* represents the passage of a data value in a message and it is drawn with a small data value icon with arrow to indicate flow direction.

4.3 Causality-flow and Timethreads

The role-based model as described so far does not provide enough behaviour information such that the operation of individual role teams and complete systems may be understood. Some behaviour cues can be inferred from the structure of the role teams, the naming of the elements, and the role wires; however, it may be difficult to piece together the operation of the system from these cues alone. There are two approaches to adding the missing

information: a component-based approach would describe the behaviour of the individual roles and role teams (perhaps using state machines); an end-to-end behaviour approach would describe the aggregate behaviour of groups of components as they cooperate.

This thesis takes the second approach and uses the concept of causality-flow [14] and the timethreads notation [10] for causality-flow to document the behaviour through a role-based model. Causality-flow refers to a sequence of causally connected activities that occur through the architectural elements of a system. Two activities are causally connected if the completion of one activity results in the second activity occurring. There is implied cause-effect machinery that results in the second activity occurring after the completion of the first, but causality-flow is not concerned with the details of how the cause-effect machinery operates. For example, one object may be responsible for maintaining a data state and informing another object when that data state changes; the second object may be responsible for displaying the data state. Causality-flow would link the activities of updating the data state, informing of the data state change, and displaying the result. Causality-flow does not concern the message exchange patterns nor the control-flow divisions among the objects involved.

Causality-flow is very similar to the concept of playing concrete scenarios in Responsibility Driven Design [7]. Concrete scenarios are played through a set of cards by selecting cards in order as the objects they represent discharge their responsibilities in the scenario. The collaboration information on the cards act as cause-effect links between the responsibilities of the objects involved in the scenario. Playing concrete scenarios is not concerned with the detailed message exchange patterns between objects. A problem with CRC cards is that there is not a convenient recording mechanism for the scenarios and designers may lose or forget valuable information.

Timethreads are a notation for causality-flow [14]. Timethreads are discussed briefly in Chapter 2 Section 2.6 and the complete timethreads notation used in this thesis appears in Appendix A. Figure 42 illustrates how timethreads are used with the role-based model. Intra-team behaviour is described by overlaying a timethread on an individual role team. The timethread traces a path through the elements of the role team that follows the causal flow

of activities performed, e.g., the activation of role methods and the updating of role attributes. Inter-team behaviour is described by a timethread that crosses possibly many role teams. Inter-team timethreads trace the cause-effect relationships between teams and provide a way of describing the end-to-end behaviour of a system. Timethreads over role models provide in one diagram a specification of behaviour sequencing as it relates to the architecture of a system. Notations which separate this information require the designer to reconstruct timethreads mentally from information that exists in two, or more, places.

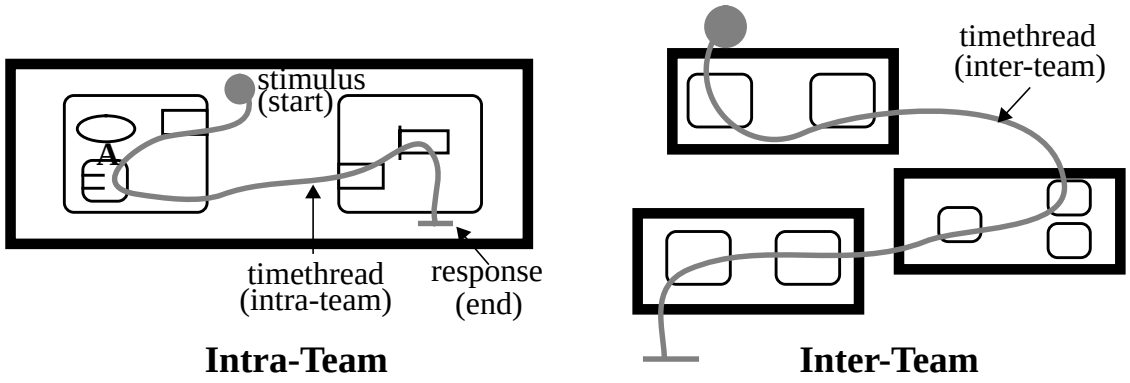

Figure 42: Timethreads through the Role-based Model

Roles wires are typically removed when timethreads are used because (1) wires tend to clutter the diagrams, and (2) the wires imply control-flow allocation which is not a concern of timethreads. During forward engineering, role wires may be developed from a role plus timethreads model. During reverse-engineering, role wires are an intermediate step towards reconstructing timethreads. This thesis emphasizes role architectures and timethreads with wiring issues as a secondary concern, however, some examples of role wirings are given.

It is easy to formalize the flow path allow a timethread using a formalism like Petri nets [43] or LOTOS [30]. It is less easy to formalize the relationship of the flow path to an underlying architecture because of the variety of possible interpretations at the detailed level. For example, the meaning of the timethread through object **A** in Figure 42 is could mean that the object's value is read or that the object is updated. This thesis uses textual annotations to indicate the proper interpretations when details are required. This has the advantage of keeping the notations and the diagrams built with the notations less cluttered. It has the disadvantage that the diagrams cannot stand independent of a textual description. It is be-

lieved, however, that this is true of any description of software architecture, including source code.

4.4 Example: Dependency Mechanism of Model/View/ Controller (MVC)

The Model/View/Controller (MVC) paradigm [41] of Smalltalk [22] divides an application into three categories of objects: Model objects maintain the application's data, Controller objects interact with the user through input devices, and View objects display the state of a Model object in different visual formats. Figure 43 provides an example of three views displaying the state of one model.

The MVC paradigm is effective because it separates the application state from the user interface which promotes reuse of these independent parts. A dependency mechanism is used by MVC applications to ensure that View objects reflect the current state of the Model objects.

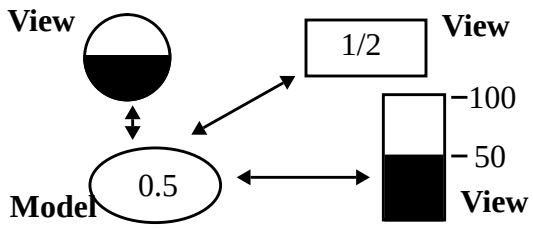

Figure 43: Models and Views

The dependency mechanism may be represented as a role team as in Figure 44. The **Dependency** role team consists of **Source** and **Dependent** roles. Model objects play the **Source** role and View objects play the **Dependent** role.

The **Dependent** role is drawn as shadowed to specify that there may be multiple **Dependent** roles in one **Dependency** team. The Dependent role is said to be replicated because a single **Source** role may have many dependents (e.g., a single Model object may have many different Views, see Figure 43).[2] The **Source** role provides a **setValue:** interface method and a **getValue** interface method for the **value** data attribute. The **Source** role has

a private method called **changed:**. The **Dependent** role provides an **update:** method. (Section 4.4.1 describes the labelling conventions that are used).

Figure 44 illustrates a timethread pattern that expresses the overall intent of the Dependency role team. The intra-team timethread is given a label (**setValue**) so that it may be referred to in other diagrams. The timethread is also is labelled with a stimulus, to describe the event that causes the timethread, and a response, to describe the result of the timethread. A stimulus from outside the **Dependency** team triggers the need to change the **value** attribute of the **Source**. This results in a timethread instance which, when completed, results in all **Dependents** of the **Source** being notified of the change (the response).

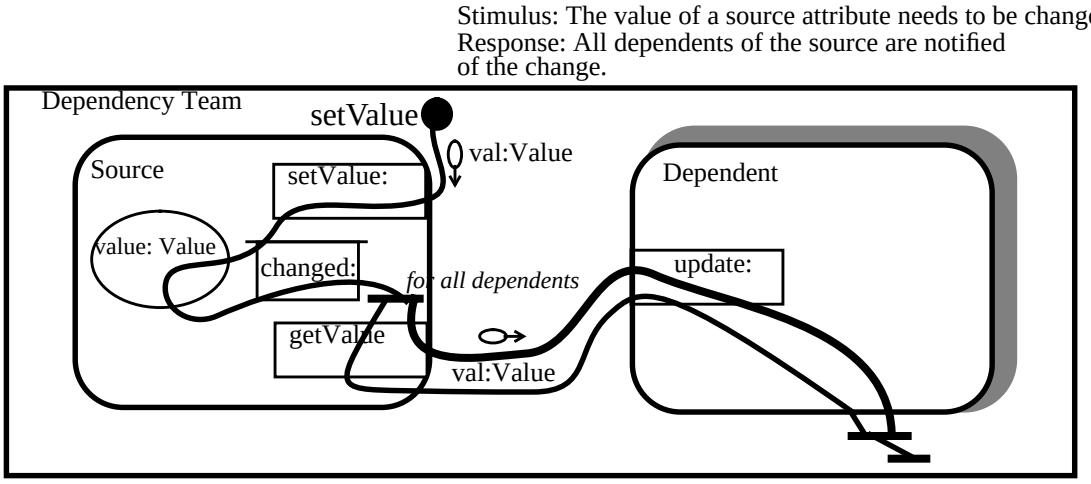

Figure 44: Behaviour of the Dependency Team

The path of the timethread specifies how the response is achieved through the ordered discharge of role activities. The thread activates the **setValue:** method and the data value **value** is updated with the contents of the data flow **val**. From there the timethread activates the **changed:** method which ensures that the **getValue:** method is activated for each **Dependent**. There is an and-fork along the timethread path just before the entry to the **getValue:** method. The and-fork is annotated with *for all Dependents* to imply that there is a path out of the and-fork for each **Dependent**. The order in which the paths are taken is not important and is left open, in fact all the paths could be taken concurrently or in some arbitrary

[2.] Replication in relation to wiring issues in discussed in Appendix C.

order. Following the **getValue:** method, each timethread path results in a **Dependent** being updated through an activation of its **update:** method. The data value **val** flows along the timethread to the **update:** method. From here the separate timethread paths synchronize at an and-join. When all the paths arrive at the and-join the overall timethread completes.

The role-based model provides a way of capturing design abstractions that are dispersed in code. For example, the methods of the Source role may not be found in the definition of a single class but are likely to be spread along several classes in a class hierarchy. In Smalltalk the equivalent of the **changed:** method is found in the top-most class called Object so all classes inherit the ability to maintain and inform dependents. The **setValue:** and **getValue** methods usually belong to a concrete subclass because they are tied to the particular **Data** that an object maintains. The timethread pattern is achieved by a fine-grained message interaction protocol between the **Source** and **Dependent** objects. It includes an **update:** message from **changed:** for each **Dependent**, and a **getValue** message from the **Dependents**.

4.4.1 Labelling Conventions

Smalltalk labelling conventions are used for the role-based model. The names of methods, objects, and data flows begin with lower case letters; role teams and roles begin with upper case letters. References to construction materials that act as templates for architectural elements begin with capital letters, e.g, the reference to the data type **Value**. The words of a compound name are separated by capitalizing the first letter of each word after the first word (e.g., **getValue**). A colon in a method name represents the need for a parameter to the method (i.e., the need for a particular element in a message). For example, the method **setValue:** requires one parameter, but **getValue** requires no parameters. An instance of a role team or an instance of an object playing a role are labelled by adding a lower case indefinite article (*a* or *an*) to the role team or role name. For example, an object instance flow labelled **aSource** would refer to an object instance playing the **Source** role.

4.5 Example: Model/View/Controller Role Team

The Model/View/Controller paradigm itself may be modelled as a role team that includes the Dependency team. Figure 45 presents the role team organization for the **MVC** team.

The role team has three roles called **Model**, **View**, and **Controller**; the **View** and **Controller** roles are replicated because a **Model** may have many View/Controller pairs. The **Dependency** role team is drawn as nested inside of the **MVC** role team. **Models** are equivalenced to **Sources** and **Views** are equivalenced to **Dependents**: one object plays the **Model** and **Source** roles and another object plays the **View** and **Dependent** roles.

Roles have obligations as implied by their methods. In Figure 45, the **Controller** role has a **userInput** method for receiving input events from the user and an **update:** method that is activated when its user prompts (e.g., menus) need updating. The **View** role has a private **display** method for redisplaying its representation of the **Model's** data. The **Model** role has a **data value** attribute, represented as such because the nature of the data is outside of the application domain of this example. The **Model** also has a set of methods for accessing the data value. The ellipis are used to say that there may be many possible interface methods provided to access the data attribute.

In Figure 45 the stimulus that launches the **userInput** thread is an input event in an area of the screen occupied by the **Controller**; the final response is that the **View** and **Controller** are updated to reflect the current state of the **Model**. The **userInput** thread satisfies the invariant that **Views** and **Controllers** reflect the state of the **Model**.

The timethread specifies the following typical sequence of activations (the numbers next to the timethread in Figure 45 represent activities that are performed): (1) the user performs an input operation (e.g., mouse click); (2) the **Controller** displays a menu; (3) the **Controller** fields the user's menu selection; (4) the **Controller** routes the user's menu selection to the appropriate method of the **Model**; (5) the **Model** activities the **setValue** timethread through the **Dependency** team, shown by the thread stub named **setValue** in the **Dependency** team; (6) the **View** displays a new representation of the **Model**; and (7) the **Controller** is updated if needed, e.g., the **Controller** may change how the menus are displayed.

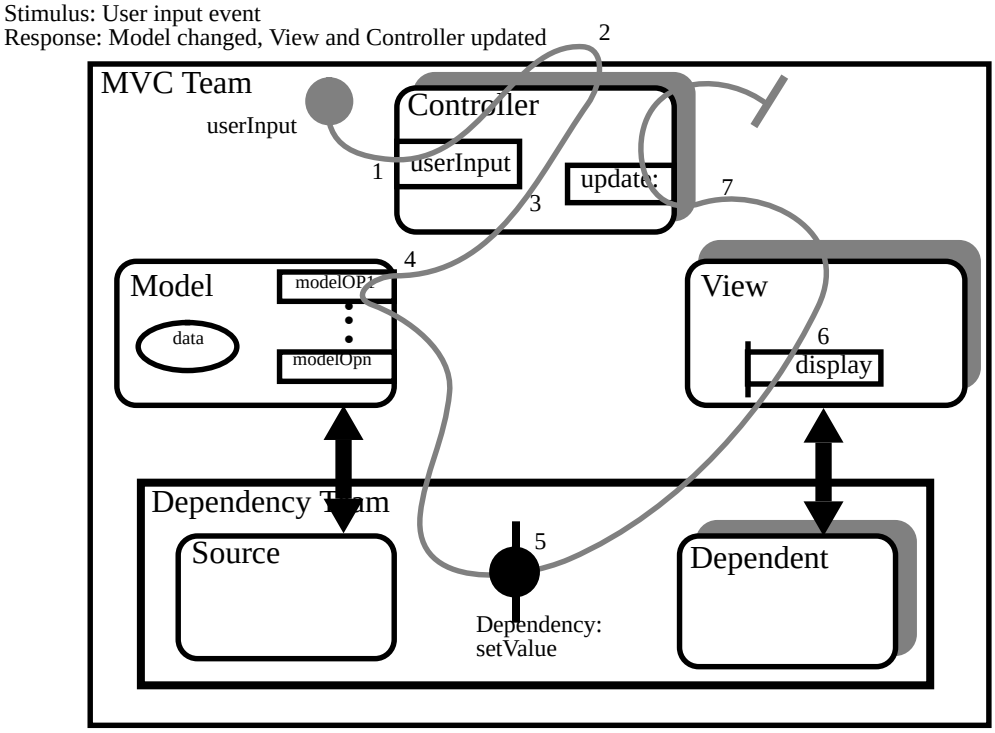

Figure 45: Model /View /Controller Team Behaviour Diagram

The purpose of behaviour diagrams is to represent the typical causal flow patterns. Such patterns are replayed many times throughout a system and define a vocabulary for understanding it. The **userInput** timethread through the **MVC** team is one example; it may have many subtle variations. For example, the **Controller** may forward some menu requests directly to the **View** if they do not effect the **Model**, such as a menu selection to change the colour of a **View's** image; or the **Model** may not activate the **setValue** thread of the **Dependency** role team if the menu selection does not cause a change to its **data** attribute.

4.6 Extending the Meta-Model

This section explains how the role-based model is derived from the meta-model of Chapter 3.

Figure 46 extends the concepts of the meta-model presented in Chapter 3 and the result is the role-based model. The concepts of the basic meta-model are shaded, and the exten-

sions to the meta-model are at the bottom of the figure. The role-based model adds the following new concepts: role teams, roles (both component-bound roles and placeholder roles), role methods (both interface role methods and private role methods), role wires, objects, and output ports. Figure 47 is a summary of the visual notation to support role-based modelling and is presented here as a reference card for the notations used in this thesis.

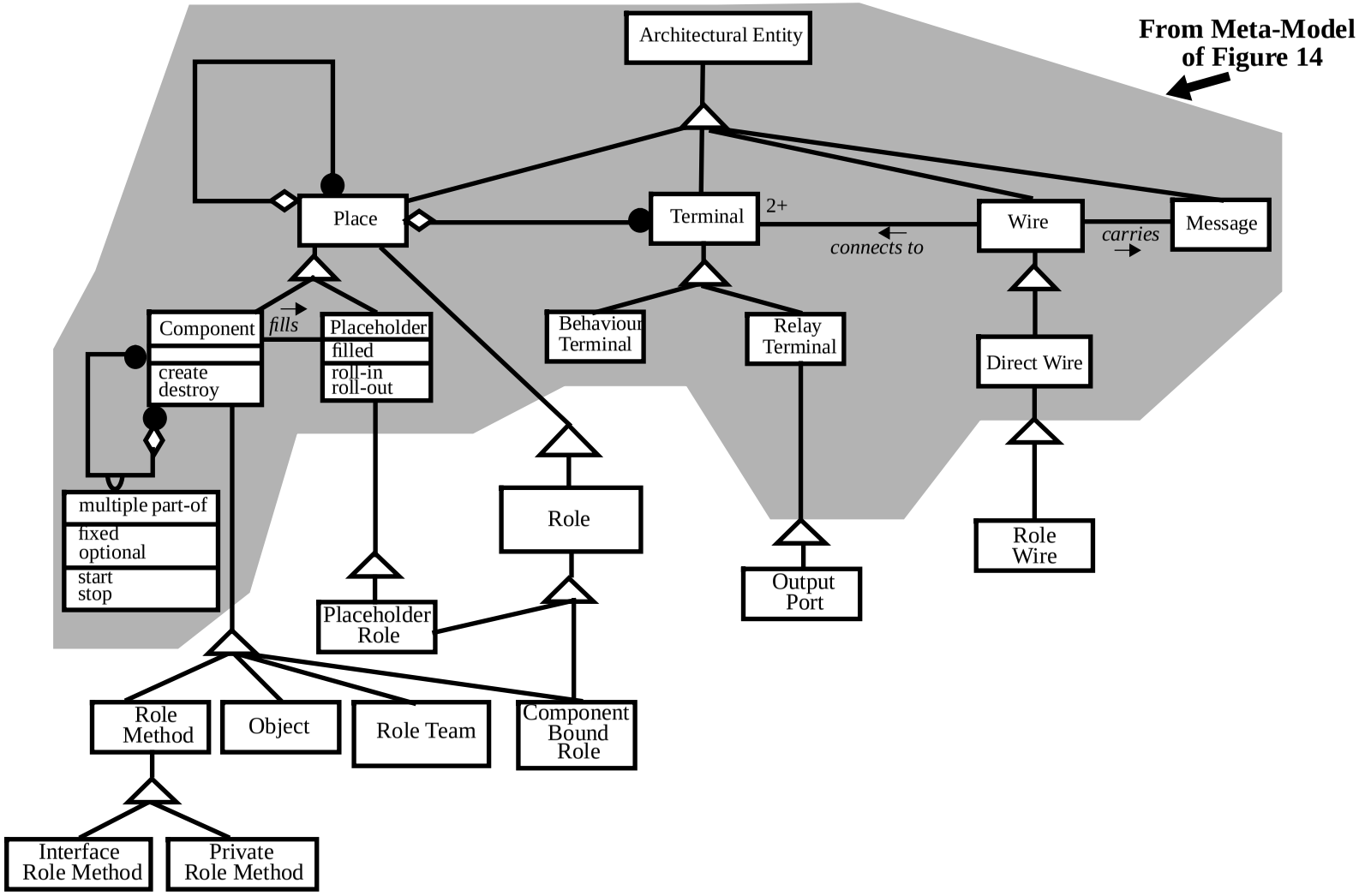

Figure 46: Extending the Meta-Model Concepts for the Role-based Model

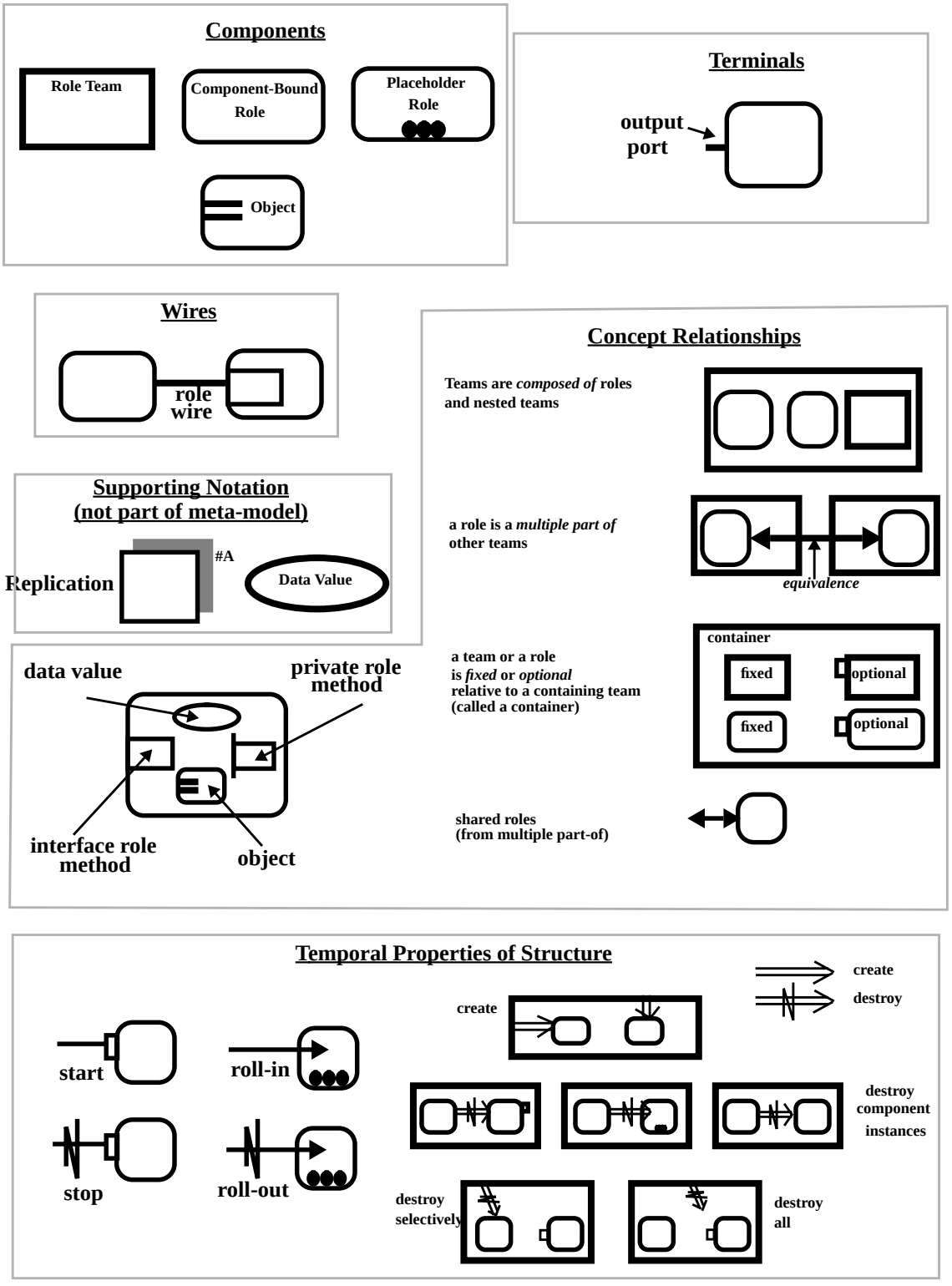

Figure 47: Summary of Notation for Role-based Model

Role teams, roles, objects, and role methods were introduced Section 4.2. Figure 46 specifies that these concepts are derived from the component concept of the meta-model. Therefore, the components of the role-based model may be recursively nested. This thesis does not constrain the nesting relationships that are possible among the different kinds of components, but there appear to be common nesting relationships. In Figure 48 container components are drawn on the right, and nested components on the left; the arcs represent the common nesting relationships. The relationships from object to role team and from role method to role team are highlighted. This is to emphasize that these relationships should be common but do not appear extensively in the case study of the thesis. The reason is that the case study applies the role-based model to an existing system in which role teams are not explicit components. Therefore, role teams do not appear in the implementation as components with their own internal state and behaviour (i.e., interface methods and private data). The author did not make use of these nesting relationships in the case study (Chapter 5) because doing so would distance the role model from the implementation.

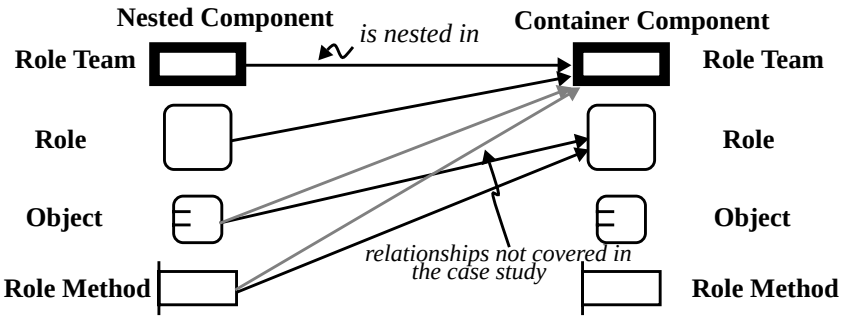

Figure 48: Common Nesting Relationships

The components of the role-based model inherit other properties: they may be involved in multiple-part of relationships, their operation in a given place may be optional, and they may be created and destroyed. Figure 49 illustrates how common these properties appear to be across the different kinds of components in the role-based model. This figure is speculative based on the case study and on the author's experience with object-oriented programming. The figure may be seen as a guideline for designing with role-based concepts and for what one might reasonably expect to find in programs.

Referring to Figure 49, role teams are unlikely be involved in multiple part-of relation-

ships because they are relatively large components and therefore less likely to be shared. Roles exhibit all of the properties. Objects when appearing explicitly as attributes of roles may be involved in multiple part-of relationships (more on this in Section 4.7.1) and may be created and destroyed. Object attributes and role methods are optional by nature, and therefore are not annotated as being optional. For example, a method is only activated whenever it is needed, so sending a message is like “starting” a method, and the exit from a method is like “stopping” it; and object attributes represent state information of an object playing a role and it is commonplace for such state information to be absent or empty. It is possible to dynamically create and destroy object attributes and role methods. Dynamic methods are possible in some languages, e.g., Smalltalk, but the concept seems to be rarely used. The sharing of methods among object instances is ubiquitous in object-oriented programs because of inheritance and polymorphism. This common form of method sharing is not represented in the role-based model; class-based diagrams can represent this information more succinctly.

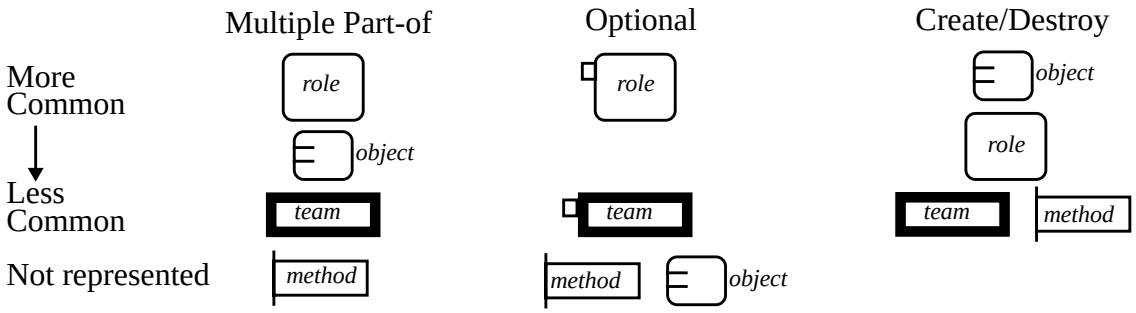

Figure 49: Common Properties of Components

Concepts of the role-based model (Figure 46) that have not yet been introduced include placeholder roles, component-bound roles, and output ports. Placeholder roles and component roles are discussed in Section 4.7 which deals with issues of dynamic structure. *Output ports* are relay terminals for outgoing messages from a role; they have no behaviour and act only as points for connecting wires. Output ports are implicit in the examples presented in Section 4.4 and Section 4.5, and are explained further in Section 4.8.

4.7 Dynamic Structure

Figure 50 extracts a fragment from the meta-model that addresses dynamic structure in the

role-based model. The right side of the figure shows some of the visual notations that represent the concepts on the left of the figure.

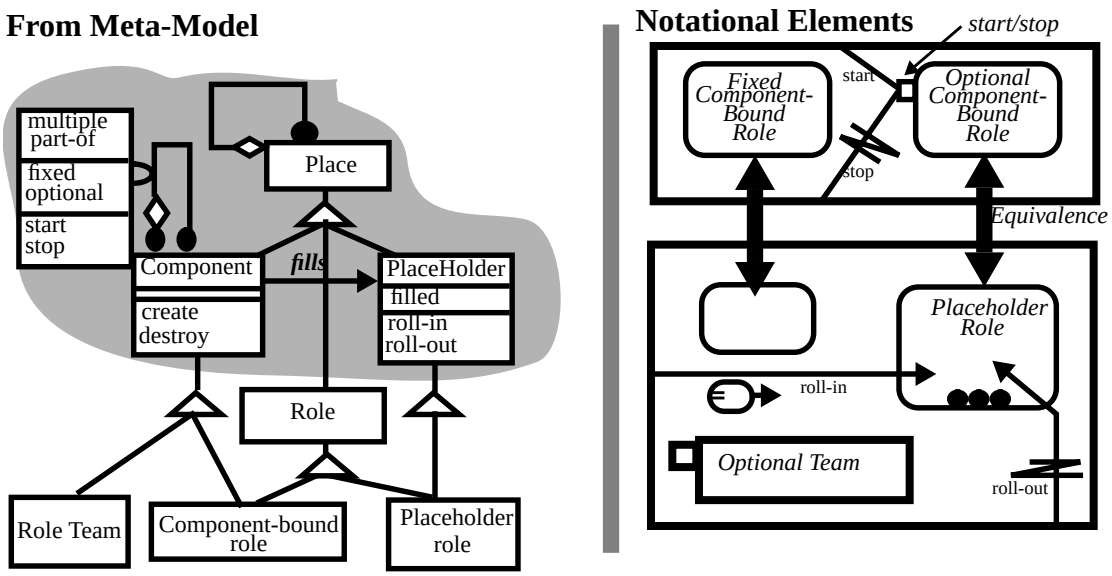

Figure 50: Dynamic Structure Relationships

A ***role*** is a place in a role archiecture that is nested in a role team; it derives from the place concept of the meta-model. A role is played by objects. Component-bound roles and placeholder roles are refinements to the general role concept. A ***placeholder role*** also derives from the placeholder concept of the meta-model, note that multiple inheritance is here as a modelling technique. A placeholder is drawn as a box with rounded corners and is annotated with rollers internal to it. Objects fill placeholder roles dynamically and different objects may fill a placeholder at different times. The objects that fill a placeholder may be playing roles in other teams or may be the attributes of roles (see below). Figure 50 provides an example of the roll-in operation which fills a placeholder and the roll-out operation which empties a placeholder.

A ***component-bound role*** derives from the role concept and component concept of the meta-model; it is drawn as a box with rounded corners. Component-bound roles are nested in role teams. The term component-bound role is meant to imply that the object that plays this kind of role does so the operational lifetime of the role. An object begins to play a fixed component-bound role (Figure 50) when the containing context (role team) of the role be-

gins its operational lifetime. An object begins to play an optional component-bound role sometime after the containing context (role team) of the role begins its operational lifetime, and the object may stop playing an optional component-bound role before the containing context of the role ends its operational lifetime. An object starts to play an optional component-bound role as a result of a start operation, and stops playing an optional component-bound role as a result of a stop operation; see Figure 50 for an example of a start and a stop operation.

A component-bound role differs from a placeholder role because only one object may ever play a component-bound role, but many different objects may fill a placeholder role through time. An object architecture view may blur this distinction because at the level of objects and object references there may be dynamic restructuring needed to establish an object in any role. For example, a fixed component-bound role is modelled as a static component that is present and operational in its containing role team for the operational lifetime of that team; but in reality there may be an exchange of identifiers at runtime in order to establish an object in the role. The same sort of dynamic behaviour may occur in the implementation when optional component-bound roles are started. One could imagine a model that consists of a set of role teams that are made up entirely of placeholder roles, and a set of objects from which role participants are selected dynamically as needed. Such a model could be built, but it would obscure the temporal constraints between roles and containing teams, and among roles in different teams, that can be specified when the distinction between role types is made. Also, such a model would not distinguish between initialization phases of components and steady-state operation. It is believed that the resulting models would be more complicated as a result. The approach taken here is to separate component initialization and operation into different diagrams; Section 4.8.4 provides an example of a role team initialization diagram.

The elements of the role-based model that address dynamic structure are illustrated in the case study of Chapter 5.

4.7.1 Explicit Objects in the Role-based Model

There are two cases when objects appear explicitly in role-based diagrams: (1) as role at-

tributes that represent state information maintained by the role; and (2) as instance flows that represent the passing of the identifiers of objects that are playing roles in a model. Implicitly objects drive the role-based model by playing roles. Instance flows represent the exchange of object identifiers that is needed in the implementation to achieve the role-based model. The exchange of the identifiers of objects that are playing roles occurs most often when teams are being initialized.

When objects appear explicitly in an architectural diagram as role attributes they may be involved in equivalence relationships. Figure 51 illustrates the three cases that can occur: (A) a role equivalenced to a role means that the same object will play both roles (perhaps simultaneously); (B) an object attribute equivalenced to a role means that the role will be played by the objects it is equivalenced to; (C) an object attribute equivalenced to another object attribute means that the same object is an attribute of both roles.

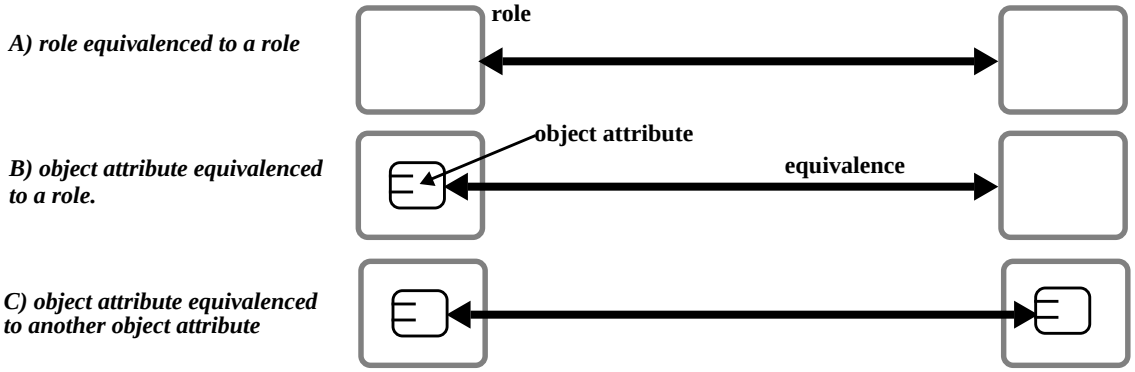

Figure 51: Objects in Equivalence Relationships

Figure 52 illustrates the temporal properties of object attributes for the destroy operation (the semantics of create for object attributes is the same as for components of the metamodel). Cases **A**, **B**, and **C** illustrate that destroying an object, whether that object is an attribute of a role or is currently playing a role, causes the destroy operation to be propagated across the equivalence relationships that join to the object. Case **D** illustrates that the destroy operation is not propagated to the containing role when an object attribute of that role is destroyed. The rationale for not propagating the destroy operation in this case is that object attributes model state information of a role and it is common place for such state information to be absent, i.e., object attributes are by their nature are optional components.

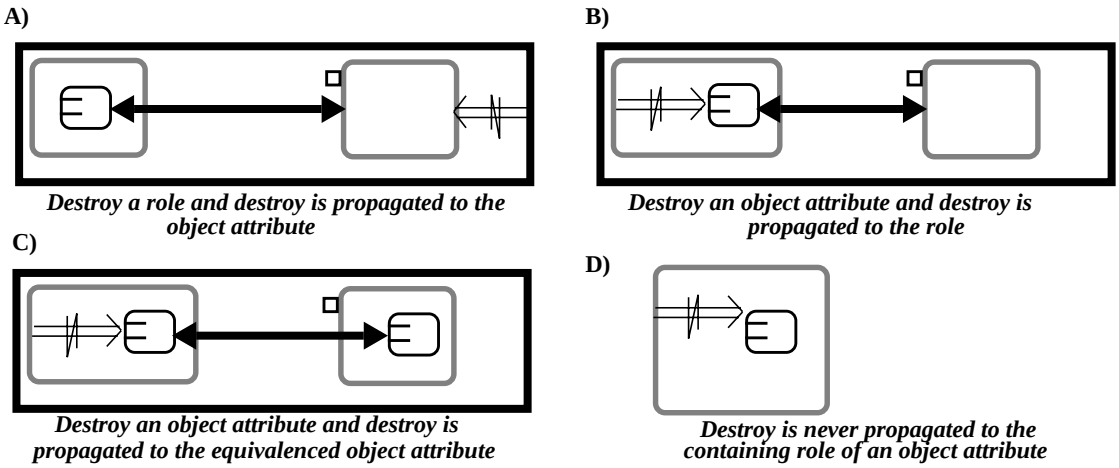

Figure 52: Temporal Properties of Object Attribute-Create and Destroy

4.8 Wiring Issues

This section contains detailed issues related to wiring and may be skipped on first reading.

4.8.1 Role Team Wiring Diagram

Figure 53 is an example of a role wiring diagram for the **Dependency** role team. A wiring diagram has control flow implications. For example, the wiring of Figure 53 implies that the **changed:** method sends a message that activates the **update:** method of the **Dependent** role, and that the **update:** method sends a message that activates the **getValue** method of the **Source** role.

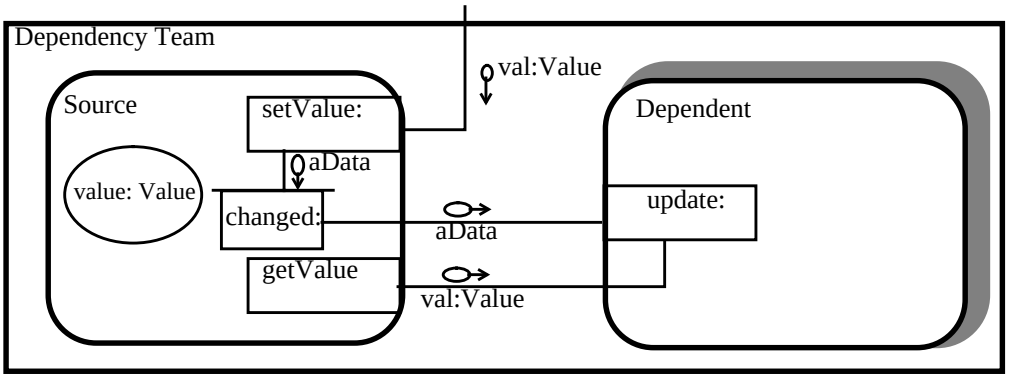

Figure 53: Role Wiring for the Dependency Team

Causality-flow is different from control-flow. Causal flow follows the major activities as they are discharged, and control flow follows the explicit messaging sequences in a system. Figure 54 illustrates the control flow through the Dependency team when there are two Dependents. The sequence is much more detailed than the causal flow sequence. In particular, **changed:** calls **update:** which calls **getValue**, there is then a return to **update:** and the actual work of updating is performed.

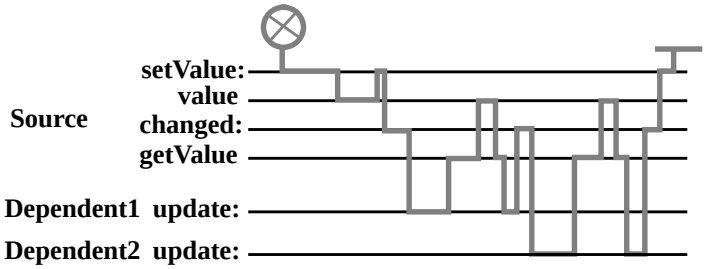

Figure 54: Control-flow of the Dependency Role Team

4.8.2 Role wires and Object References

Role wires are achieved in the implementation by dynamic object references established through the exchange of object identifiers. Role wires are a more abstract concept than object references. A role wire appears as a direct connection between roles, when in fact there may be a exchange of object identifiers needed to achieve the connection. The

exchange of identifiers may occur once during the establishment of a role team or continuously throughout its operation whenever a role wire is used. For example, consider the wire between the **changed:** method and the **update:** method in Figure 53. This wire may be established once through the exchange of object identifiers between the **Source** and **Dependents** when the role team is initialized. Alternatively, there may be a manager involved that maintains the identifiers and a **Source** could obtain the identifiers from the manager as needed.

In strictest terms, a role wire is not a communication pathway. In Chapter 1 a communication pathway is defined as a logical connection between components that may be decomposed into substructures. A communication pathway can model cases where a logical path exists between two components that is achieved through an intermediate party. Role wires are more concrete because they may not be decomposed in this manner.

4.8.3 Wires are Bound to Terminals

The meta-model states that wires are bound to terminals. This is still true in the role-based model. The role-based model introduces the concept of an ***output port***. An output port is a relay terminal; it is drawn as a T an sits on the edge of a role or a role team. An output port acts as a wiring point from inside to outside across the boundaries of a role or role team.

Figure 55 shows how output ports, role wires, and role methods combine visually. Implicit on a role method are terminals for wiring. The elongated side of the method is for connecting wires that carry incoming messages, and wires that carry outgoing messages are wired to the remaining three sides.

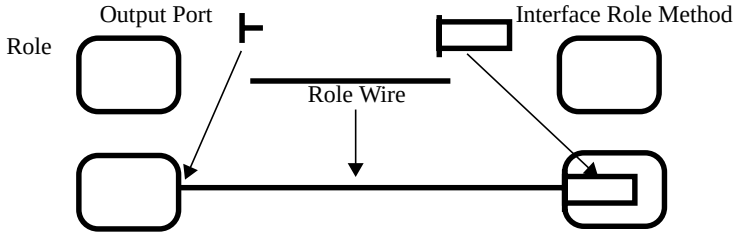

Figure 55: Combining Output Ports, Role Wires, and Interface Methods

An output port must be compatible with an interface role method for the two to be bound by a wire as in Figure 55. Recall that the compatibility of terminals is deferred in the basic meta-model. In the role-based model an output port is compatible with an interface method if all messages sent from the output port have a given number of contents and the types of the contents are as expected by an interface method. The issue of data typing for the contents of message is not pursued further here because this is an aspect of system construction that is not explicitly addressed by the role-based model.

4.8.4 Role Team Initialization Diagram

It is during role team initialization that objects are created and identifiers exchanged to create the context for the operation of a team. Initialization diagrams are not important for understanding the steady-state operation of a system, but they illustrate how a role-based model is achieved using the building blocks of the implementation technology.

Figure 56 is a team initialization diagram for the **MVC** team. The creation arcs represent the creation of object instances to play the roles of the team. The creation of the **MVC** team creates a **Model** object, which creates one or more **View** objects, and the **View** objects each create an associated **Controller** object. The nested **Dependency** role team is implicitly created when its container (the **MVC** role team) is created. A number of interface methods are used to exchange object identifiers, e.g., the **on:** method of the **View**. The **addDependent:** method of the **Model** is provided to initialize the nested **Dependency** team with **View** objects that play the **Dependent** role. The equivalence arcs make the sharing of objects across roles explicit.

The initialization diagrams separate creation (shown by the creation arc) from object initialization (shown by the exchange of identifiers through method invocations). Object-oriented programming languages allow these two operations to be combined in class methods.

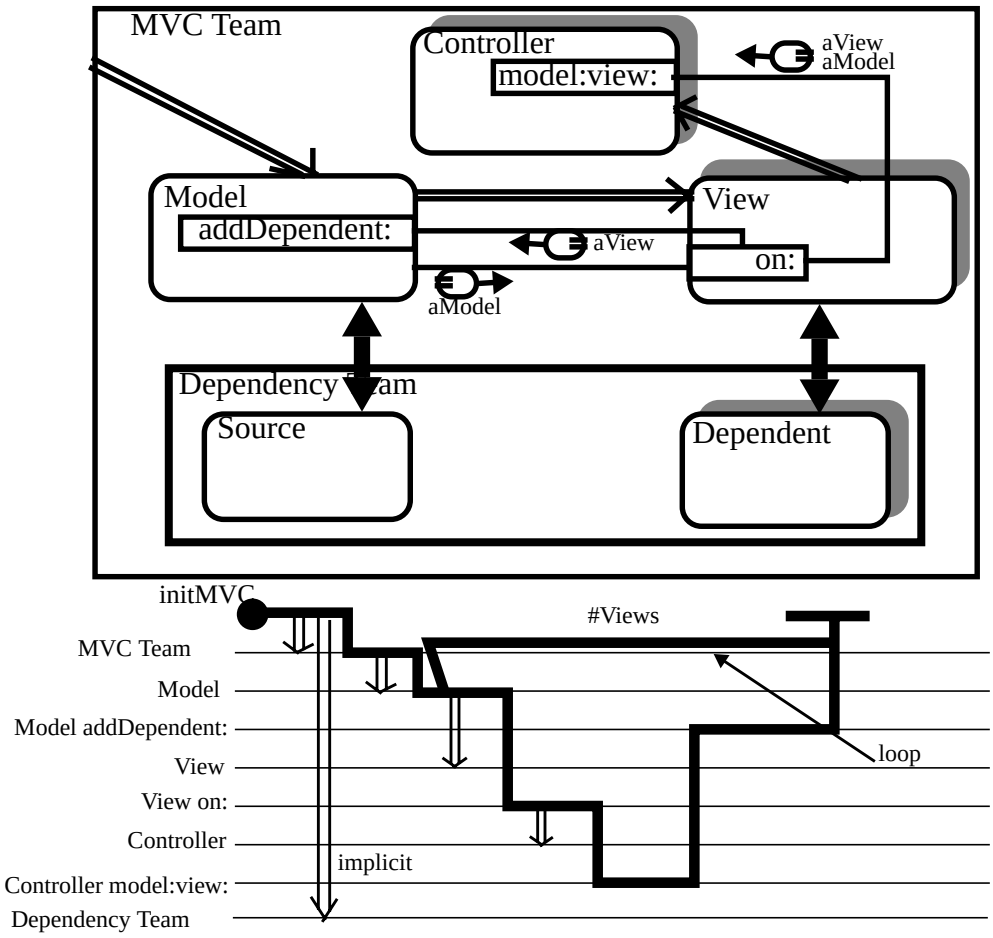

Figure 56: Model /View /Controller Role Team Initialization Diagram

The initialization timethread is drawn in linear form over a timing diagram at the bottom of Figure 56. Creation arcs between the axis represent points where object and role team instances are created. There is a loop in the timethread from its end to the point where **View** objects are created; the result is a repeated pattern for creating multiple View/Controller pairs.

4.9 Summary

The role-based model is more static than an object architecture: role teams are often repeated throughout a system with different object participants, and role wires are connection between roles. A factoring into roles is a projection from a relatively dynamic model to a

more static one [13], that is analogous to the projection of the grouping plane from the wiring plane discussed in Figure 28 in Chapter 3. Conceptually, there is a relatively static architectural model in terms of role teams and roles, and a relatively dynamic one in terms of objects and object references with object role playing as the link between the two. The static nature of the role-based model enables one to reason about the architecture of an object-oriented system in more general terms and also provides a more static substrate for playing causal flow patterns.

Chapter 5: Case Study: HotDraw

In this chapter, the role-based model is applied to the HotDraw object-oriented framework. HotDraw is written in Smalltalk [22] and was designed using standard object-oriented techniques: role teams are not explicit in the design of HotDraw. The author discovered a clean role architecture that expresses (in his opinion) the intent of the program. The fact that a clean role architecture underlies the program indicates (in the author's opinion) that object-oriented designers implicitly think in these terms, even if they don't express their designs this way.

5.1 Background on HotDraw

HotDraw is an object-oriented framework for creating semantic graphic editors for 2D drawings. It can be used to build editors for specialized drawings, such as hardware schematics or software design diagrams. Figure 57 shows the user interface of HotDraw before it has been customized. Figures are drawn on a drawing canvas by selecting an appropriate tool from a tool palette. The initial set of figures in the tool palette area include: lines, arrows, rectangles, circles, and text. Some other tools in the tool palette include the selection tool, the scroll tool, and the eraser tool. When selected, a tool becomes the current tool and is highlighted in the tool palette area. The current tool controls the interpretation of user actions (e.g., mouse clicks) in the drawing area. A figure or figures may be selected with the mouse using the selection tool. A selected figure has handles displayed on it that may be used to change its attributes, e.g., size and colour. A group of figures represents a drawing which may be saved to and loaded from disk.

The case study will examine the governing role teams of HotDraw which manage user interaction and display in the tool palette and the drawing canvas areas. (There are a number of other teams not documented here which manage the manipulation of figures, e.g., resizing a group of figures, maintaining connections between joined figures, and animation). Emphasis is placed on showing how the teams work together to achieve the end-to-end behaviour of the framework.

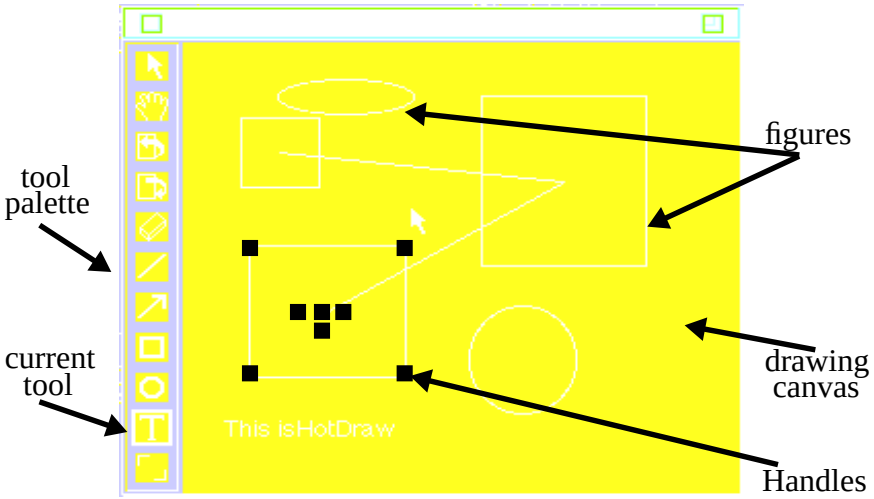

Figure 57: HotDraw A Semantic Editor for 2D Graphics

HotDraw itself is built on the user interface framework of Smalltalk [22]. The user interface framework of Smalltalk has its own set of teams which are inherited by HotDraw and are assumed as implied detail for the purposes here. For example, Smalltalk provides the basic control loop to manage the coordinated dispatch of mouse and keyboard events to the appropriate windows of a Smalltalk application.

5.2 The MVC Triads of HotDraw

Like most Smalltalk Graphical User Interface (GUI) applications, the organizing architectural principle behind HotDraw is the Model/View/Controller (MVC) paradigm. In Section 4.5, a role-based model of the MVC paradigm is presented and timethreads are used to describe the overall intent of the role team. Recall that the MVC team consists of three roles: a Controller which manages user interaction and menu display; a Model which maintains

the application data; and a View which displays that data. A nested Dependency role team (see Section 4.4) is used to ensure that objects playing the View role(s) are notified of changes to the data of the Model.

HotDraw consists of three MVC teams: the **EditorMVC** team, the **DrawingMVC** team, and the **ToolMVC** team. These three teams are refinements to the basic MVC team presented in Section 4.5. Figure 58 shows the "is-a" relationships between these teams. The specific MVC teams of HotDraw refine both the structural aspects and the behavioural aspects of the basic MVC team.

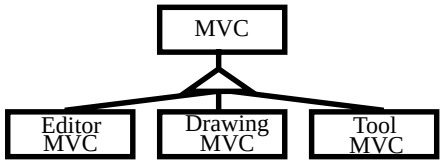

Figure 58: Refinement of MVC Teams in HotDraw

The goals or objectives of the three MVC teams are:

1. The *EditorMVC* team is responsible for loading and saving drawings to and from the file system and for ensuring that the current drawing is displayed. A drawing is a set a figures that may be treated as a unit.
2. The *DrawingMVC* team is responsible for user interaction and the display of figures in the drawing canvas area.
3. The *ToolMVC* team is responsible for user interaction and the display of the current tool in the tool palette area.

Figure 59 is a role-based diagram of the three governing MVC teams of HotDraw. The diagram shows only the team/role organization of the MVC teams: details like role wirings of individual teams are omitted. Diagrams like this are useful because they present a high-level view of system organization.

Each team consists of three roles: a **Controller**, a **Model**, and a **View**. Note that the **Controller** and **View** roles are not replicated. This represents a refinement to the structure

of the basic MVC team, that is appropriate because there are not multiple views to maintain for each model in HotDraw, but only one. Certain roles are shared across the teams; e.g, the **DrawingController** role is shared between the **EditorMVC** and **DrawingMVC** teams, the **DrawingView** role is shared between the **EditorMVC** and **DrawingMVC** teams, and the **DrawingEditor** role is shared between the **EditorMVC** and **ToolMVC** teams. All the roles are fixed except for the **CurrentDrawing** role of the **DrawingMVC** team which is modelled as a placeholder role. This allows the current drawing that HotDraw is editing to be changed dynamically at runtime. All the other roles are fixed component-bound roles and the role teams are fixed; therefore, the objects which play these roles must be created when the system starts, and they must play the roles for their entire composite lifetimes. Also, the objects which play roles that are equivalenced must play those roles at the same time.

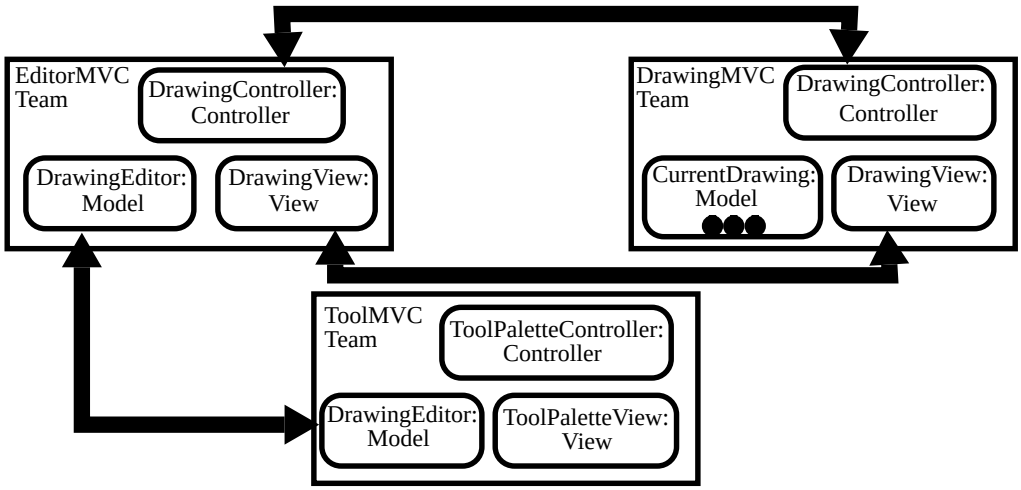

Figure 59: The Governing MVC Teams of HotDraw

Missing, and assumed as implied, from Figure 59 are the nested Dependency teams of each of the MVC teams. Each MVC team actually has a structure similar to that shown in Figure 60 which shows a more complete picture of the EditorMVC team. This structure implies that the View role is dependent on the Model role in each of the MVC teams in Figure 59.

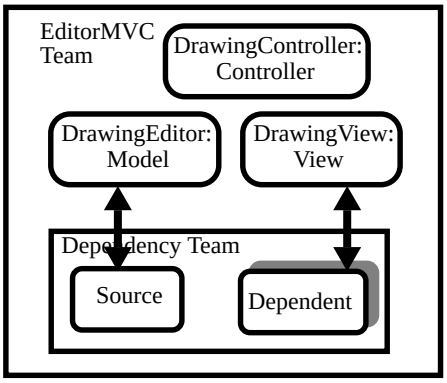

Figure 60: Nested Dependency Teams are Assumed

Where possible the names given to elements of the role-based model of HotDraw are borrowed from the source code of the framework. For example, **DrawingEditor** is the name given to a class in HotDraw. It is an instance of the DrawingEditor class that is assumed to play the **DrawingEditor** role in various teams. The names of role methods and object attributes are also kept consistent with method and instance variable names in the source code. This should ease the mapping from the role architectural model to the source code of HotDraw

The individual MVC teams are described in isolation and then in combination.

5.3 The EditorMVC Team

The *EditorMVC* team is responsible for loading and saving drawings to and from the file system and marking the drawing that is the current drawing. A drawing is a set a figures that may be treated as a unit.

Figure 61 is an intra-team behaviour diagram of the **EditorMVC** team. There are three roles which refine the roles of the basic MVC team:

The *DrawingController* is a kind of controller role (note the syntactic convention *DrawingController: Controller* which specifies this). The **DrawingController** manages mouse input in the drawing canvas and has an interface method, called **yellowButtonActivity**, [1] for this purpose. The yellowButtonActivity method signals to the controller that it is to display a

menu to the user. The user may prompt this activity by selecting a particular mouse button on a multi-button mouse or on a single button mouse by pressing a control key in conjunction a mouse click.

The *DrawingEditor* is a kind of model role. It maintains a group of drawing object attributes. The **Drawings** objects are modelled as persistent within the **DrawingEditor** but they may be stored on disk and not loaded. The role method **loadDrawing:** causes a new drawing to become the current drawing, and **saveDrawing** saves a representation of the drawing to disk.

The *DrawingView* is a kind of view role. It is dependent on the **Drawings** objects of the **DrawingEditor**. It has a private **redisplayAll** method that is activated when a new **Drawing** is selected as the current drawing (HotDraw can display one drawing at a time). When a new current drawing is selected, the entire drawing canvas must be erased and the contents of the new current **Drawing** displayed.

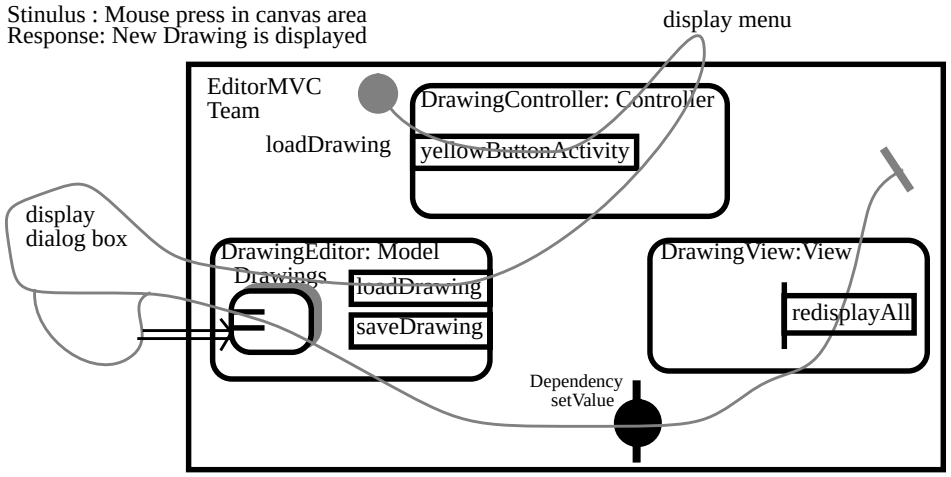

Figure 61: The EditorMVC Team of HotDraw-—Intra-Team Behaviour

The *loadDrawing* timethread of Figure 61 begins with a mouse press in the drawing canvas and results in a new drawing being displayed by the **DrawingView**. The path of the timethread is a follows: a mouse press in the canvas area activates the **yellowButtonActivity** method; the **DrawingController** then presents a menu to the user; the user selects the load option from the menu; the **DrawingController** activates the **loadDrawing:** method of the **DrawingEditor**; the **DrawingEditor** displays a dialogue box asking the user for the

[1.] The name of this method has historical roots in the development of Smalltalk. It is left unchanged because it appears this way in the source code of HotDraw.

name of the drawing to load; the user enters a drawing name; at this point there is a fork in the path: if the drawing does not exist a new one is created, but if the drawing already exists it becomes the current drawing; the data maintained by the **DrawingEditor** has changed at this point which causes the **setValue** timethread through the **Dependency** team between the **DrawingEditor** and **DrawingView** to be activated, shown by the stub thread labelled **Dependency:setValue**; and finally, the **DrawingView** updates its entire display through its private **redisplayAll** method.

The **loadDrawing** timethread of Figure 61 is a variant of the standard timethread of the MVC team. Both timethreads follow the same general path: user input prompts controller, controller prompts user, user responds, controller informs model, model updates itself, the dependency team is launched, and the view updates its display (if necessary). It is after the user inputs a specific menu item that the **loadDrawing** timethread deviates from the standard MVC one. It is after this point that a specific method of the model (**loadDrawing**) is activated. Another deviation occurs when the **DrawingEditor** prompts the user with the dialogue box. Knowledge of the general patterns provides a context for understanding the deviations.

5.4 The DrawingMVC Team

The *DrawingMVC* team (Figure 62) is responsible for user interaction and the display of figures in the drawing canvas area. The roles of the DrawingMVC team are:

The *CurrentDrawing* maintains a set of **Figure** object attributes. These **Figure** objects are the model data on which the **DrawingView** is dependent. The **CurrentDrawing** role supports two interface methods **addFigure:**, which creates new **Figure** objects, and **removeFigure:** which destroys **Figure** objects.

The *DrawingView* in this context is dependent on the individual figures of the **CurrentDrawing**. When a figure is removed or added the **Dependency** team between the **CurrentDrawing** and the **DrawingView** must be activated. The **DrawingView** is responsible for redrawing the portion of the canvas that is damaged by the add or the remove operation. In the **EditorMVC** team the **DrawingView** is dependent on the **Drawings** objects, which represent a set of figures. A change to the model data in the context of **EditorMVC** team results in the redisplay of the entire canvas. In the context of the **DrawingMVC** team, a change to the model data (figures) results in a redisplay to only those portions of the canvas that are affected.

The **DrawingView** has three role attributes: a set of **SelectedFigures**, a **CopyBuffer**, and a set of **Handles**. The **SelectedFigures** represent the objects on the screen that are currently selected (figures are selected by picking them with the selection tool). These objects are the focus of subsequent operations that involve the **DrawingMVC** team. The SelectedFigures are equivalenced to the **Figures** of the **CurrentDrawing** which means that the same objects are attributes of two different roles and implies that **SelectedFigures** are some subset of **Figures**.[2] The **CopyBuffer** represents some data structure appropriate for temporarily housing copies of figures during operations like cut and copy. It is modelled as a data attribute because it is not an object which will play roles in the role-based model of the framework. The set of **Handles** represent the filled squares which appear on a figure when it is selected. The interface role methods of the **DrawingView** support the user interface operations that may be applied to the group of **SelectedFigures**.

The *DrawingController* handles user interaction. It has a **yellowButtonActivity** method for this purpose.

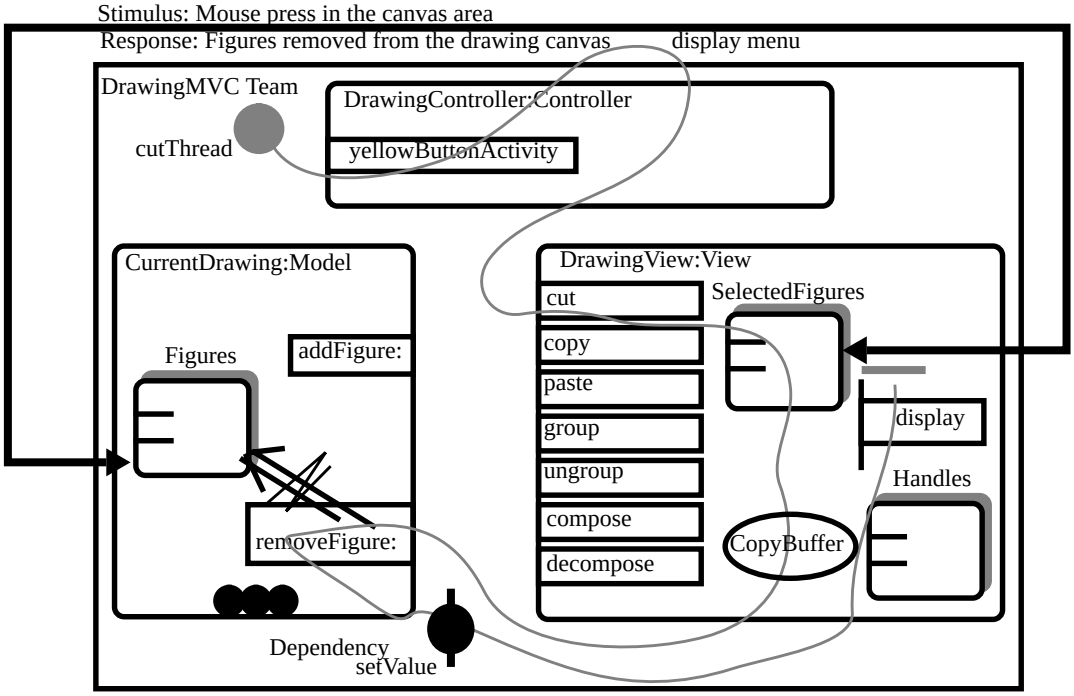

Figure 62: The DrawingMVC Team—Intra-Team Behaviour

Figure 62 illustrates a timethread path through the **DrawingMVC** team. The *cutThread*

2. Textual constraints could be added to formalize this relationship.

timethread begins with a mouse press in the canvas area and ends with the current group of selected figures removed from the drawing canvas. After the user selects the cut operation from the menu, the timethread activates the cut method of the **DrawingView**, the **SelectedFigures** are copied into the **CopyBuffer**, and the **removeFigure:** method of the **CurrentDrawing** is activated. The destroy arc from the timethread to the group of **Figures** is meant to symbolize the destruction of the figures that where selected. The equivalence joining the **Figures** to the **SelectedFigures** causes the destroy operation to propagate to the **SelectedFigures**. The result is that the **SelectedFigures** are removed from the **DrawingView** role. Destroying object attributes does not cause a destroy operation to be propagated to the containing role of those attributes (see Section 4.7.1). Destroying **Figures** of the **CurrentDrawing** causes the **setValue** thread of the nested **Dependency** team to be launched. The result is an update to the **DrawingView** to display the drawing now that some figures are gone.

Missing from the above explanation is a treatment of the **Handles** objects which should be destroyed when the figures they are associated with are destroyed. The **Handles** objects are discussed further in Section 5.8.

The timethread pattern through the **DrawingMVC** team is a common variant of the standard MVC timethread pattern. The variation is that the timethread proceeds from the Controller to the View, instead of from the Controller to the Model. The reason for this change is that the **DrawingView** maintains the group of **SelectedFigures** which are the focus of the user operations: cut, copy, paste, group, etc. Associating the user operation methods with the **DrawingView** increases cohesion because the methods are associated directly with the data they manipulate.

Another approach is to move the role attributes and the associated role methods from the **DrawingView** to the **CurrentDrawing**. This would preserve the more standard MVC timethread pattern, but would introduce non-model data to the model role and blur the distinction between the model and view roles. **SelectedFigures** and **Handles** do not represent model data because the meaning of a drawing, as expressed by a group of figures, is not changed when certain figures are selected.

The above trade-off highlights the difficulty of assigning responsibilities to roles (and objects and classes). The role-based model cannot in itself be used to choose one approach over the other. It is a useful tool, however, for highlighting the trade-offs by making the variations in structure and behaviour patterns explicit.

5.5 The ToolMVC Team

The *ToolMVC* team is responsible for user interaction and the display of the current tool in the tool palette area. Its roles are:

The *DrawingEditor* is a model role. It maintains a group of **Tools** object attributes that correspond to the list of tools in the tool palette area. The **currentTool:** role method is for setting the current tool which will interpret subsequent user operations in the tool palette area.

The *ToolPaletteView* is a view role. It displays the tools in the tool palette area which includes highlighting the currently selected tool.

The *ToolPaletteController* is a controller role. It manages user interaction in the tool palette area and has a role method, called **redButtonActivity**, for this purpose. A **redButtonActivity** is distinct from a **yellowButtonActivity** in that a **redButtonActivity** does not display a menu.

Figure 63 illustrates the *selectTool* timethread through the **ToolMVC** team. It follows the standard MVC patterns and should be self explanatory.

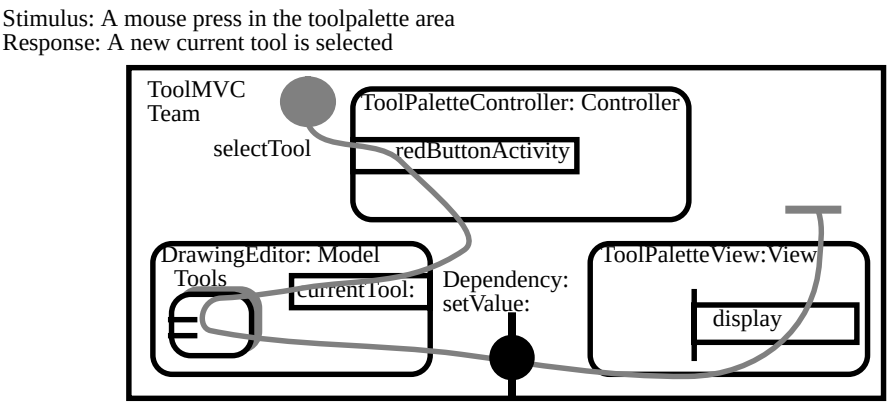

Figure 63: ToolMVC Team— Intra-Team Behaviour

The remaining sections examine how these teams (and some others) operate together.

5.6 The EditorMVC and DrawingMVC Teams

Figure 64 illustrates the relationships between the **EditorMVC** and the **DrawingMVC** teams. The **DrawingController** is the same object in both teams (i.e, they are equivalenced). The **DrawingView** roles are equivalenced. **Drawings** objects are equivalenced to the **CurrentDrawing** role. This means that the objects which play the **CurrentDrawing** role must come from the **Drawings** objects group. The **CurrentDrawing** role is not replicated; therefore, only one **Drawing** object playing this role at one time. In other words, HotDraw allows only one drawing to be edited at a time. The **DrawingView** role of the **DrawingMVC** team is responsible for rolling **Drawings** objects into and out of the **CurrentDrawing** role (as specified by the roll-out and roll-in arcs).

In the **EditorMVC** team the **DrawingView** is dependent on the **Drawings** objects, which represent a set of figures. A change to the model data in the context of **EditorMVC** team results in the redisplay of the entire canvas (the **redisplayAll** method). In the context of the **DrawingMVC** team, a change to the model data (figures) results in a redisplay to only those portions of the canvas that are affected (the **display** method). This represents a subtle distinction that can be difficult to see in the code: in the **EditorMVC** team a **Drawing** object is the model data, in the **DrawingMVC** team the contents of a **Drawing** object playing the **CurrentDrawing** role is the model data. In both cases, the view role is played by the same object, but the responsibilities of that object are different in the two cases.

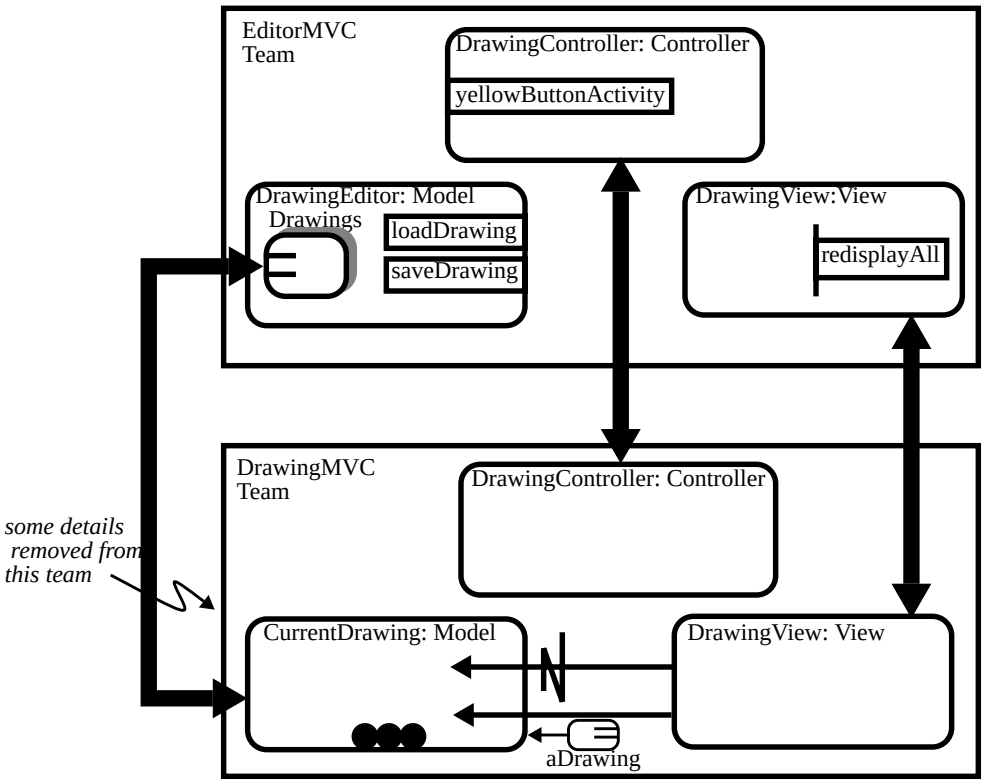

Figure 64: EditorMVC and DrawingMVC Teams

Figure 65 illustrates one possible inter-team timethread for the **EditorMVC** and **DrawingMVC** teams. The timethread is an extended version of the **loadDrawing** timethread of the **EditorMVC** team from Figure 61. The timethread shown here travels to the **DrawingMVC** team and activates the **DrawingView** role to roll-out the old **CurrentDrawing** and to roll-in a new one. The cause-effect relationship between the activation of the **Dependency** team in the **EditorMVC** team and the installation of a new **CurrentDrawing** in the **DrawingMVC** team is achieved through the shared **DrawingView** role. The timethread is drawn next to the equivalence relationship between the shared **DrawingView** roles to illustrate this cause-effect relationship. The new **CurrentDrawing** was selected through a dialogue box filled in by the user and is shown as flowing along the timethread.

After the CurrentDrawing has been rolled in, the **redisplayAll** method of the DrawingView is activated; this updates the entire display to reflect the new current drawing.

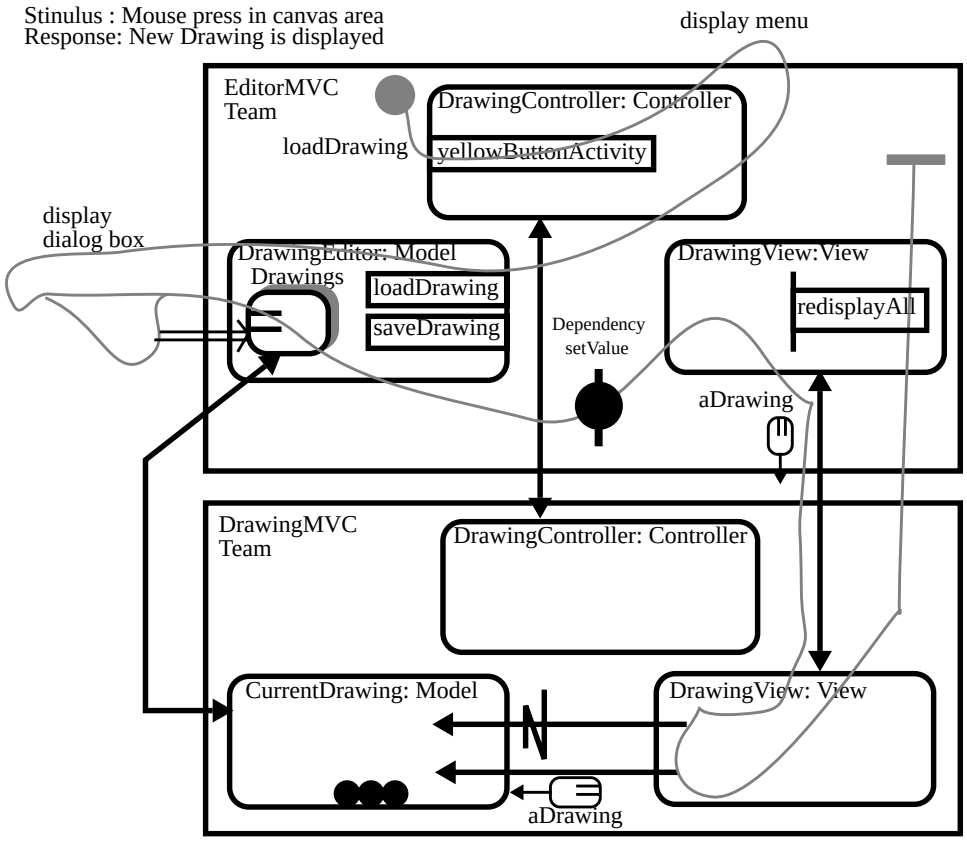

Figure 65: EditorMVC and DrawingMVC—Inter-Team Behaviour

The example of Figure 65 illustrates one way that teams can communicate: through shared roles. The **DrawingView** role is shared between the **EditorMVC** and the **DrawingMVC** teams. When the **Dependency** team between the **DrawingEditor** and the **DrawingView** in the **EditorMVC** team fires the result is to cause the **DrawingView** to take action in the **DrawingMVC** team. Timethreads allow the causal relationships between actions in different teams to be shown explicitly.

5.7 The ToolMVC Teams and MouseDown Teams

Figure 66 is an inter-team diagram for the **ToolMVC** and **MouseDown** teams. This is the first mention of **MouseDown** team. The **MouseDown** team handles mouse clicks in the drawing canvas that are fielded by the currently selected tool. The **DrawingMVC** team

(Figure 62) also fields mouse clicks in the drawing canvas, but only for mouse clicks that require a menu to be displayed.

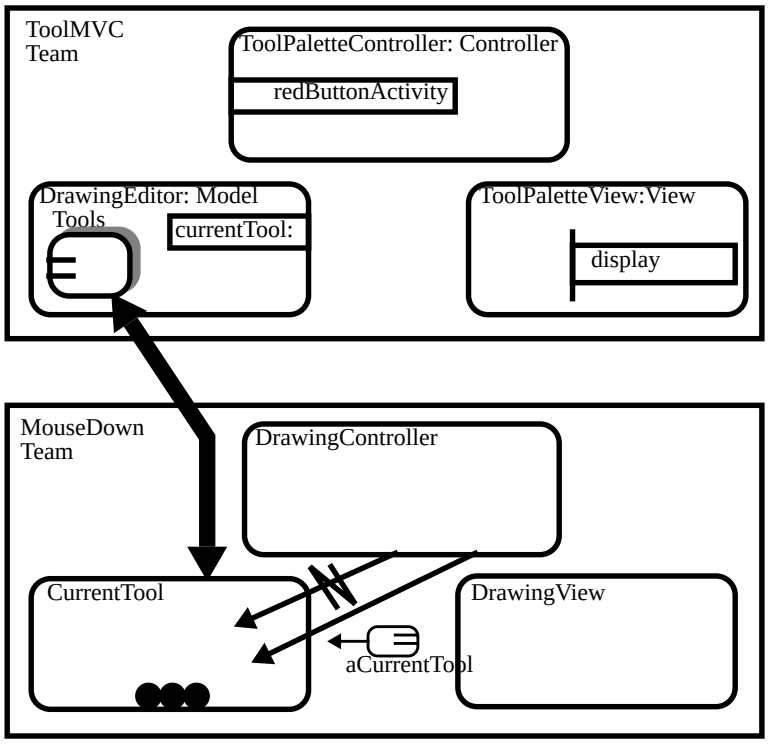

Figure 66: ToolMVC and MouseDown Teams

Figure 67 is an intra-team behaviour diagram of the **MouseDown** role team. **Mouse-Down** is not an MVC team, although the **DrawingController** and **DrawingView** roles are equivalenced to controller and view roles of the **EditorMVC** and **DrawingMVC** teams (not shown in figure). The **CurrentTool** has a press: method that is activated when a mouse click is detected in the drawing canvas. One of the private methods **pressFigure** or **pressBackground** is eventually activated depending on the position of the cursor when the mouse is clicked. The **MouseDown** team is an abstract team because it must be specialized for the behaviour of the specific tools, e.g., the selection tool.

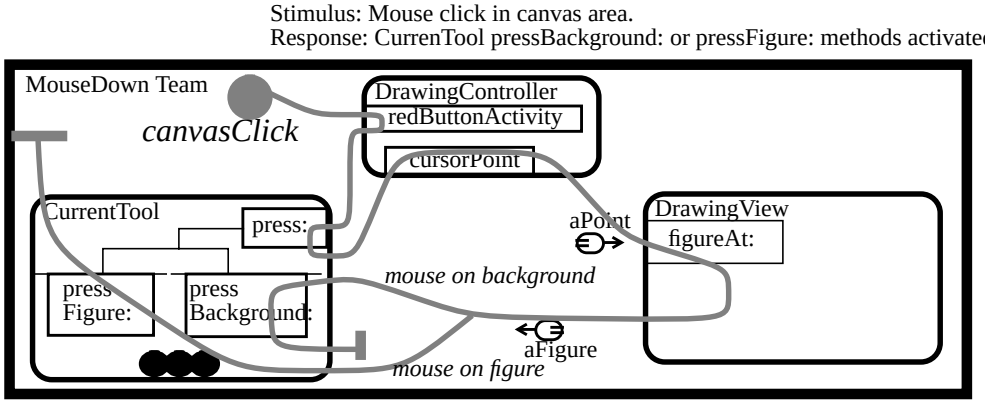

Figure 67: The MouseDown Team—Intra-Team Behaviour

Figure 68 is an inter-team behaviour diagram for the **ToolMVC** and **MouseDown** teams. The *selectTool* timethread shown is an extended view of the selectTool timethread shown in Figure 63. In this extended view the timethread travels to the **MouseDown** team after the current tool has been set in the **DrawingEditor** of the **ToolMVC** team. The **DrawingController** of the **MouseDown** team then rolls out the old **CurrentTool** and rolls in a new one. Subsequently, the **Dependency** team between the **DrawingEditor** and the **ToolPaletteView** fires, causing the **ToolPaletteView** to highlight the new current tool.

Note the similarity in structure and the similarity in the timethread patterns of Figure 65 and Figure 68. Regularity of form is an important property of systems that eases may development activities, e.g., maintenance, debugging, and re-engineering. The examples illustrate how the role-based model can expose the regularity of form (or lack thereof) in a system.

The example of this section illustrates another way in which teams may communicate: through the roles of a third team. For example, the **ToolMVC** and **MouseDown** teams have no shared roles. Yet to achieve the model the **DrawingController** must be informed of the new current tool. In the HotDraw implementation, this is achieved by a connection between the **DrawingController** and the **DrawingEditor** that occurs as a result of the **DrawingMVC** team. The role-based model abstracts from these details. The timethread segment in Figure 68 that extends from the **DrawingEditor** to the **DrawingController** says that selecting a new current tool causes the **DrawingController** to fill the **CurrentTool** placeholder with the new current tool object. The details of the cause-effect machinery to achieve this are abstract-

ed way.

One may argue using a third team to achieve inter-team communication complicates the implementation of the cause-effect relationships across teams. This is true, but this case study is modelling an existing system in which role teams are not explicit components. Making role teams explicit components would clarify the relationships between objects and thus the purpose of the references between objects. If role teams were explicit, then the **ToolMVC** team could communicate directly with the **MouseDown** team when a new current tool is selected. This would eliminate the need for communicating *through* a third team.

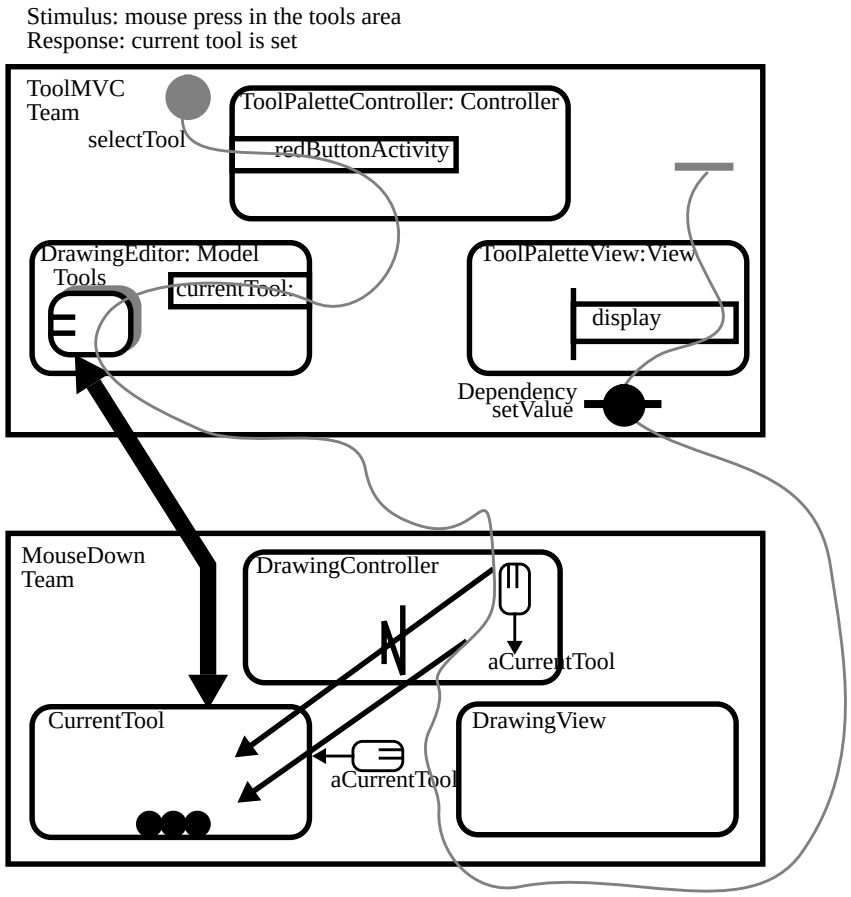

Figure 68: ToolMVC and MouseDown Teams-—Inter-Team Behaviour

Timethreads are a useful tool for reasoning about and documenting alternative behaviour patterns. For example, Figure 68 illustrates an alternative timethread pattern for the in-

ter-team behaviour of the **ToolMVC** and **MouseDown** teams. Figure 68 uses an and-fork which decouples the strict sequencing between installing a new **CurrentTool** and updating the display of the **ToolPaletteView**; the new timethread pattern allows these activities to proceed independently. The goal may be to allow for concurrency in the implementation, or just to document the logical concurrency relationships between activities. During forward engineering, representing the logical concurrency relationships in timethreads allows for a wider range of possible solutions than if an artificial and strict sequencing is imposed. During reverse-engineering the logical relationships may not be apparent if the implementation enforces a strict sequencing. In any case, several timethread diagrams may be needed which progress from more abstract (which may represent logical concurrency) to more specific (which are closer to how the implementation operates).

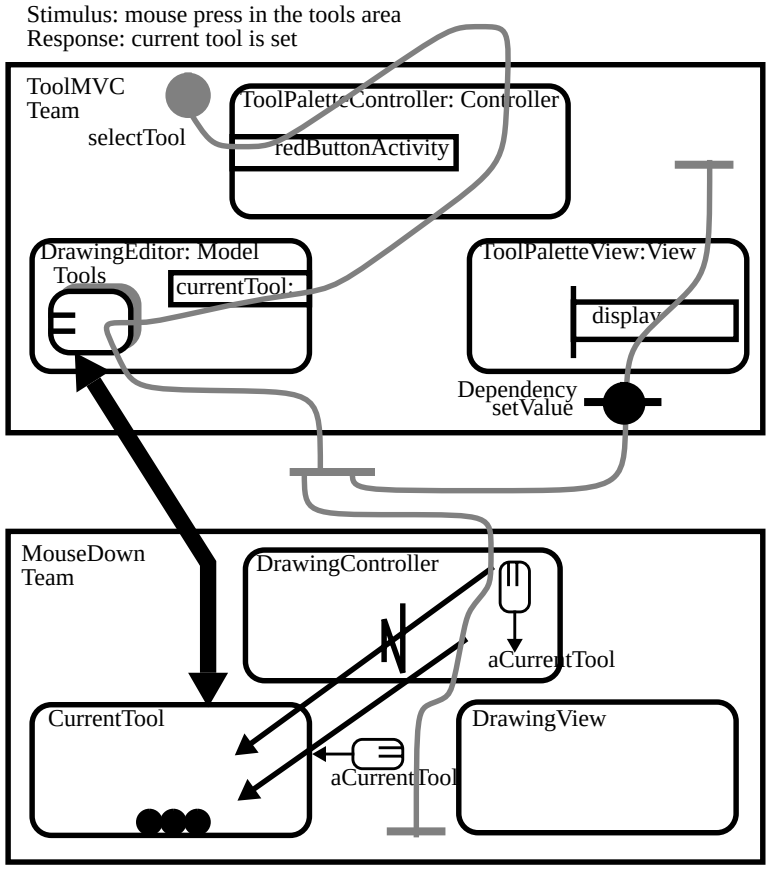

Figure 69: An Alternative Behaviour Pattern

5.8 The DrawingMVC and FigureSelection Teams

Figure 70 is an inter-team diagram of the **DrawingMVC** and **FigureSelection** teams. The **FigureSelection** team is a refinement of the **MouseDown** team that was introduced in Figure 67. It is a refinement because the **SelectionTool** role refines the **CurrentTool** role of the **MouseDown** team: the **SelectionTool** role requires more specific properties of the objects that play it than does the **CurrentTool** role (the details are not presented here). Also there is a nested team, called **HandlesFigure**, that is part of the **FigureSelection** team. The **HandlesFigure** team is replicated and optional within the context of the **FigureSelection** team. The **DrawingView** role is responsible for starting and stopping instances of the **HandlesFigure** team. A **HandlesFigure** team is started whenever a figure is selected with the mouse, and stopped if a figure is deselected with the mouse. The **HandlesFigure** team is optional because the **FigureSelection** team can operate independently of whether a figure is currently selected or not. Many details of the **FigureSelection** and **HandlesFigure** team are omitted but could be developed in separate diagrams.

The **HandlesFigure** team specifies the interaction between handles and figures necessary for operations like resizing and figure connection. It consists of two roles. The **Handles** role is replicated because a **Figure** may have many **Handles**. A destroy arc from the **HandlesFigure** team to the **Handles** says that when the **HandlesFigure** team is destroyed that the **Handles** are to be destroyed with it. Destroying a **HandlesFigure** team does not destroy the objects playing the **Figure** role, which is logical since a figure can exist without being selected.

The equivalence relationships of Figure 70 specify that **SelectedFigure** objects play the role of **Figures** in the **HandlesFigure** team and that **Handles** objects play the role of **Handles** in the **HandlesFigure** team. **SelectedFigures** objects are equivalenced to **Figures** objects. The following semantics are implied by the equivalence relationships: (1) Stopping a **HandlesFigure** team destroys it because it is not shared in any other context. This causes a destroy operation to be sent to the objects playing the **Handles** role of the **HandlesFigure** team and for the destroy operation to be propagated to the **Handles** attributes of the **DrawingView**. (2) Destroying a **Figures** object in the **Drawing** role causes the destroy operation

to be propagated to the **SelectedFigures** objects of the **DrawingRole** removing them from the **DrawingView**. The destroy operation is also propagated to the objects playing the **Figure** role of the **HandlesFigure** team. Since the **Figure** role is fixed the destroy is propagated the **HandlesFigure** team destroying it. The destroy is also propagated to the objects playing the **Handles** role and the case (1) above is repeated.

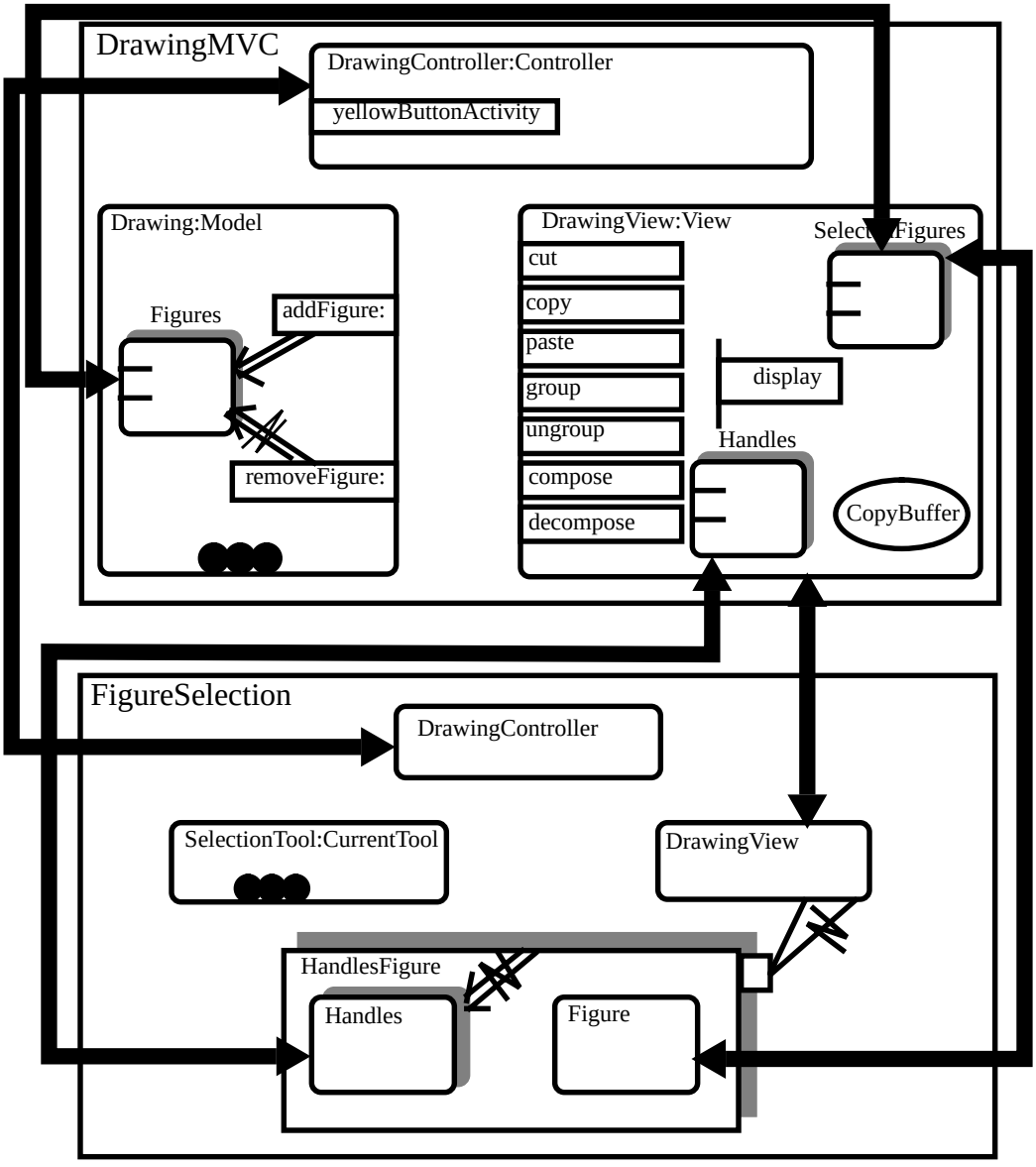

Figure 70: DrawingMVC and FigureSelection Teams—Team Wiring

An inter-team behaviour diagram could be developed but enough of the case study has been presented to illustrate the important concepts of the role-based model.

5.9 Back to the Big Picture

Figure 71 is a summary of most of the teams presented in the case study. It shows the teams, their internal structure without wiring details, and the equivalencies among the teams. Diagrams like Figure 71 are important because they expose the global structure of a system at several levels without an over-committment to details.

Figure 71 shows a causal flow pattern through the teams of Figure 71. The causal flow is driven by the following interaction scenario with the user: the user selects a new drawing to display; the user chooses to work with the selection tool; the user selects a figure of the drawing; and the user cuts the figure from the drawing. This interaction scenario drives several consequent timethreads; the timethreads were presented in the previous sections: **loadDrawing** (Section 5.6), **selectTool** (Section 5.5), **clickCanvas** (Section 5.7), and **cutThread** (Section 5.4). The **loadAndCutFigure** timethread shown in Figure 71 gives end-to-end continuity to the threads of which it is composed.

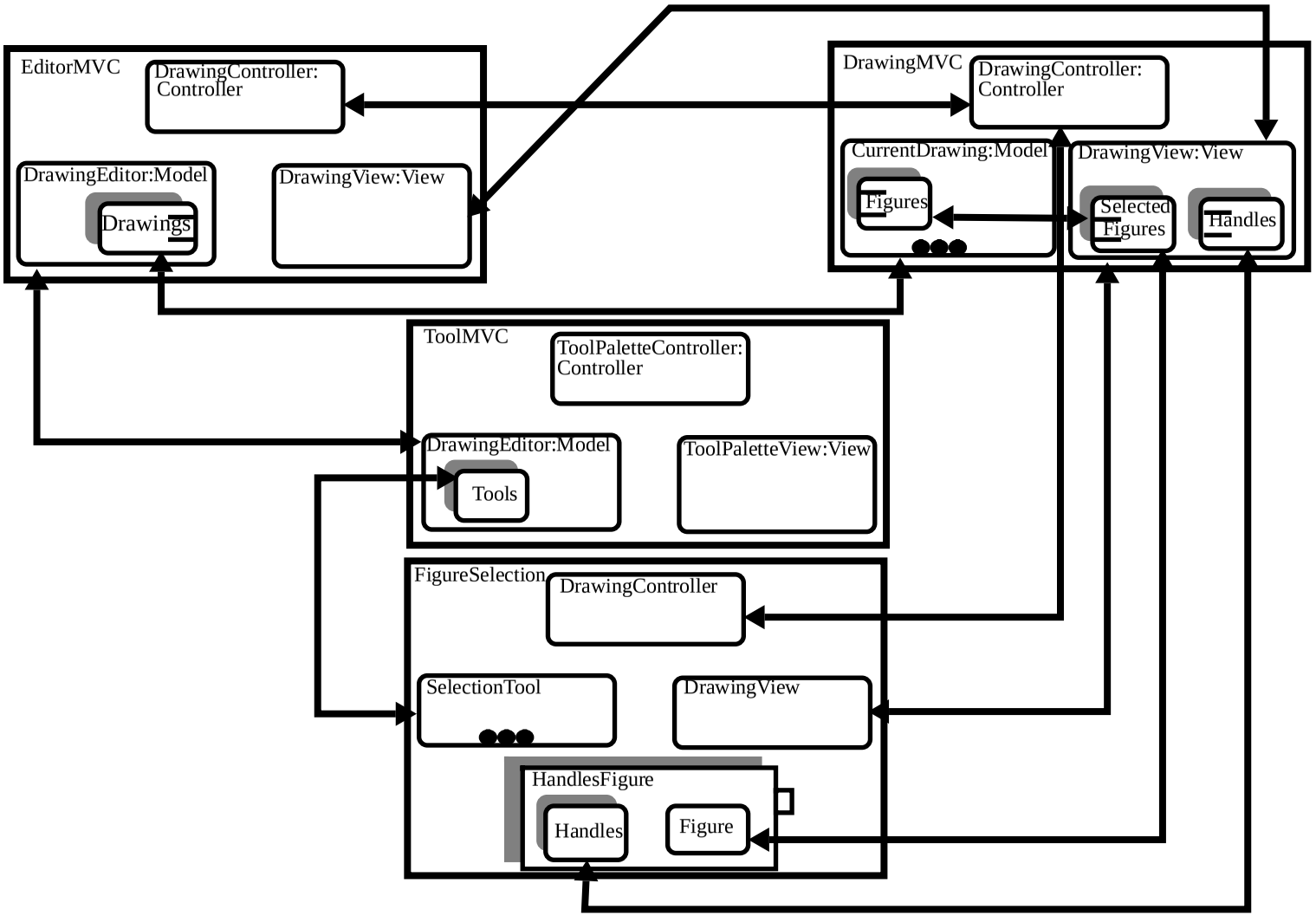

Figure 71: Summary of Teams in the HotDraw Case Study

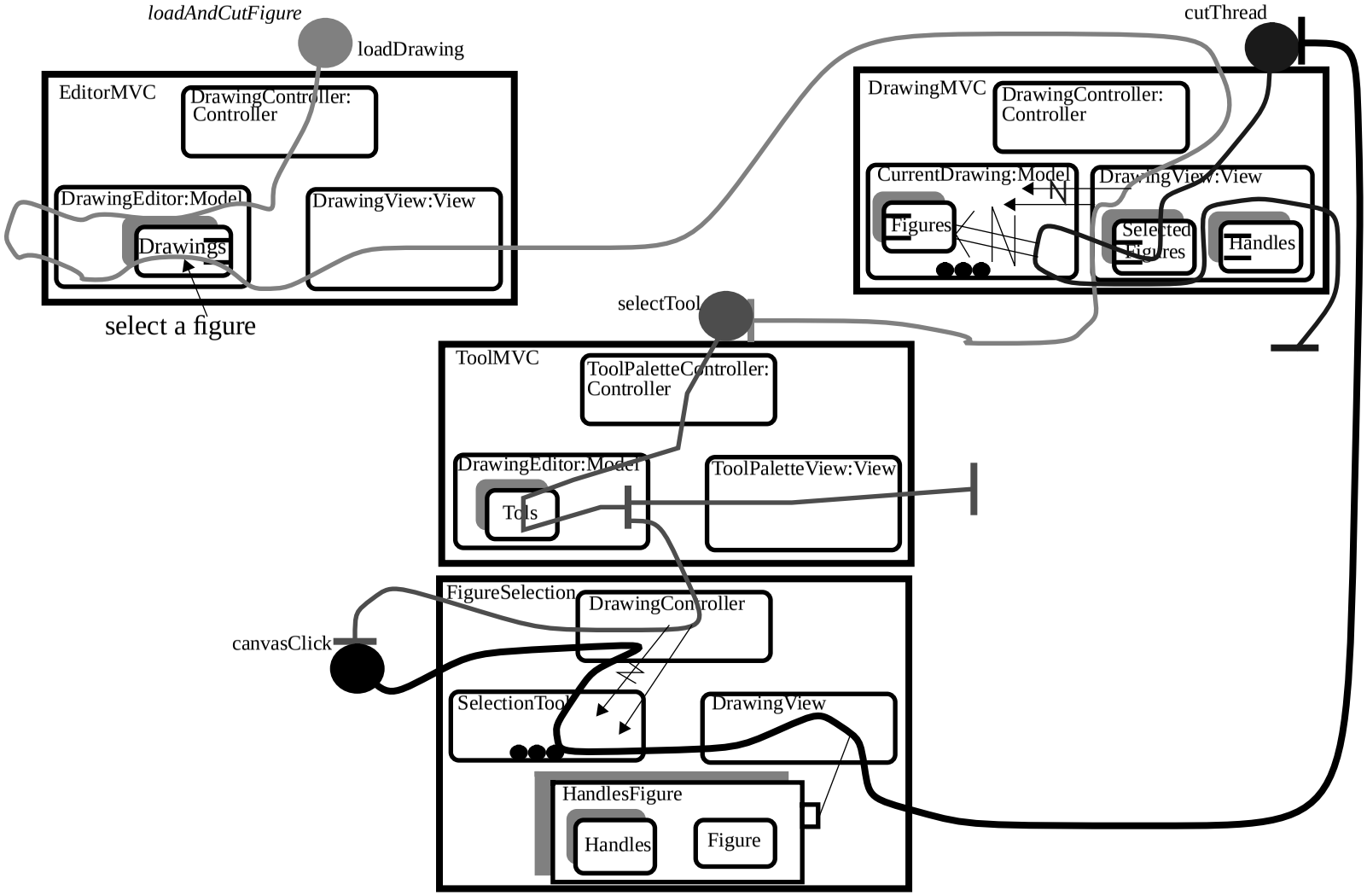

Figure 72: Causal Flow Through the HotDraw Teams

5.10 Wiring Diagrams

The section presents the role team wiring diagrams for the **EditorMVC** (Figure 74), **DrawingMVC** (Figure 73), and **ToolMVC** (Figure 75) teams. The previous timethread diagrams deferred control-flow issues, like who calls whom. The wiring diagrams here have implicit control-flow assumptions because of the directional nature of the role wires. Role wires together with timethreads provide the necessary information to infer control-flow details.

For example, Figure 73 (the **DrawingMVC** team wiring diagram) and Figure 62 (the **DrawingMVC** intra-team behaviour diagram) together imply the following: that the **DrawingController** sends the **cut** message to the **DrawingView**; the **DrawingView** then sends the **remove:** message to the **DrawingEditor**; the **DrawingEditor** invokes the dependency mechanism by sending the **changed:** message to itself; and so on until the message stack is unwound and control returns to the **DrawingController**.

The filled squares at the junction points of wires in the following figures are called routers. A *router* is a terminal detached from the edge of a place that routes messages based on their contents; they create less cluttered diagrams because they allow for shared wires. Routers are explained further in Appendix C. The remaining figures of this section are presented without further explanation.

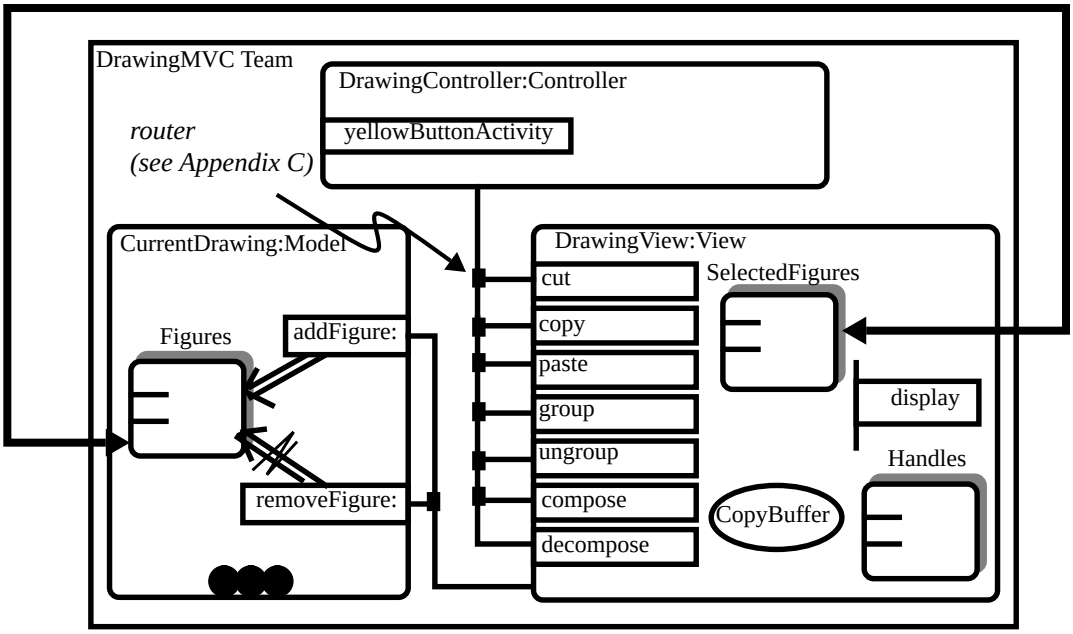

Figure 73: The DrawingMVC Team—Team Wiring

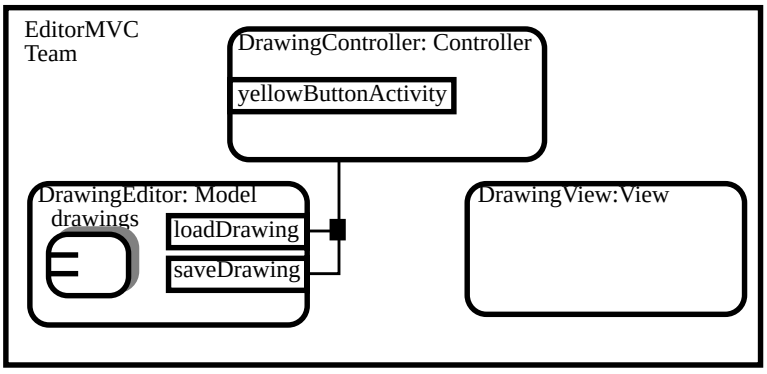

Figure 74: The EditorMVC Team of HotDraw-—Team Wiring

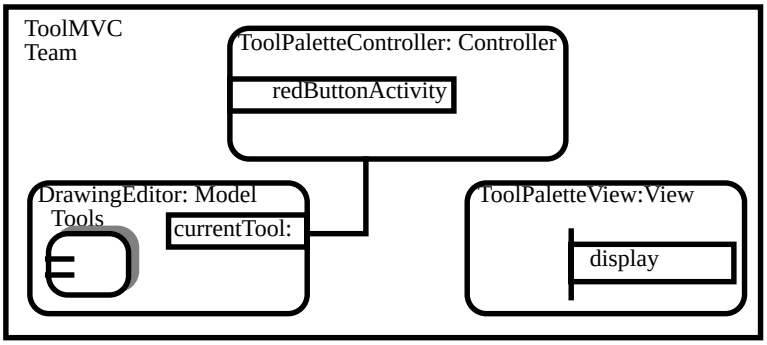

Figure 75: ToolMVC Team—Team Wiring

5.11 Summary

The case study illustrates that the role-based model is applicable to both individual teams and groups of teams. The intra-team diagrams model the architecture of individual teams and inter-team diagrams can scale up to model the architecture of a system from a global perspective. Timethreads are applicable for both intra-team and inter-team diagrams. The inter-team diagrams make use of timethread stubs which allow one timethread to refer to another without showing all the details.

The role-based model divides a system up into a teams. The model is successful if the teams can be understood in isolation and if the global architecture of a system can be understood when the teams are combined. It is the author’s opinion that the role-based model, and timethreads, are successful. However, true success will only come from applying the model to a larger number of case studies.

Chapter 6: Conclusions

6.1 Research Summary

The following problem is posed in Chapter 1.

> *The problem of this thesis is how to model and represent the following aspects of an object-oriented system's architecture: (1) the architecture of individual groups, (2) the architecture of a system as composed of groups, (3) how the participants play their roles in multiple groups and the dependencies across groups, (4) the typical causal-flow patterns though the system, and (5) dynamic structure.*

The research addressed the above problem by developing a generalized meta-model for the architecture of software systems (Chapter 3). The meta-model uses the following concepts: places, components, placeholders, terminals, wires, and messages. The architecture of a system is described as a set of places (components and placeholders) and the wires between the places to support communication. Wires are joined to places at terminals. A place may be recursively nested and is a black-box because it interacts with other places only by message passing through its terminals. Multiple part-of means that the same component instance may be in multiple places (containers) perhaps at different times or at the same time. The temporal properties of a component in a container may be defined as fixed or optional and placeholders allow multiple components to come and go from a container through time. The composite lifetime of a component may be determined by considering its temporal properties in all the places the component operates in. Dynamic structure includes compo-

nent creation, destruction, and “movement”. Creating a component acquires necessary system resources and gives the component a unique identity; destroying a component reclaims system resources and marks the component’s identifier as invalid. A component “moves” when it is rolled into or out-of a placeholder.

The role-based model (Chapter 4) extends the meta-model with the following concepts: role teams, roles, objects, role methods, role wires, and output ports. Role teams are templates for collaboration groups and roles are templates for objects that participate in collaboration groups. Objects appear explicitly in the role-based model as attributes of roles (object attributes) and as instance flows along wires and timethreads. Object attributes model state information; object instance flows along wires and timethreads may represent the exchange of identifiers or the *movement* of objects to different places. Objects appear implicitly in a role-based model by playing roles and are the source of computational activity. Role methods are primitive components of roles that perform activities; activities are the responsibilities that objects must perform when they play a role. Role methods are activated by sending messages over role wires. An output port is a wiring point on the edge of a role.

The role-based model is applicable to systems with dynamic structure but is more static than the basic object architecture of object-oriented systems as characterized by objects and object references. Role wires are fixed connections between roles that may be achieved through dynamic means in the implementation (e.g., identifier passing). Placeholder roles are fixed in a wiring diagram, but the objects that fill a placeholder may come and go through time. The more static nature of the role-based model provides a better basis for reasoning about the architectural organization of a system and for playing behaviour patterns.

A visual notation, based on the MachineCharts notation of Buhr [11] is used, to express system architecture in the role-based model. The visual notation is suggestive and expresses the major concepts of the role-based model in visual form. Some aspects of system behaviour cannot be expressed with the role-based model alone. In particular, timethreads [10] are used as means of capturing the major causal-flow patterns through a role team and across multiple role teams. Emphasis is placed on capturing the major causal flow patterns

over developing complete specifications.

The role-based model addresses the needs outlined above in the following ways:

1. *The architecture of an individual group* is expressed as a role team. A role team is composed of roles and, perhaps, nested role teams. An intra-team behaviour diagram is a role team diagram with a timethread pattern overlaid on it (wires are typically removed from such diagrams). Informal prose are used to describe the relationship between the timethread pattern and the role team diagram. A role team wiring diagram shows how the roles of a team are wired together.

2. *The architecture of a system as composed of groups* is represented using diagrams that show multiple coordinated role teams. These diagrams show the structure of a system to several levels with details of wiring and interface methods removed. The diagrams show the temporal properties of role teams and roles, i.e., whether they are optional or fixed in a given place; and placeholder roles are shown.

3. *How the participants play their roles in multiple groups and the dependencies across groups* are represented using equivalence relationships in diagrams that show multiple coordinated role teams. Equivalence relationships are drawn between places (i.e., roles and object attributes) that contain the same object at runtime. An equivalence relationship alone does not specify the temporal properties of role playing, i.e., when an object plays a role relative to its other roles and relative to the role playing of other objects with which it participates. Some temporal properties of role playing may be determined by examining the structure of a system, the temporal properties of the roles and role teams, and the equivalence relationships between roles. Timethreads may also be used to add extra temporal information, for example, a timethread may specify that an object is operating in one place before it is rolled into a placeholder some where else.

4. *The typical causal-flow patterns though the system* are represented using Buhr's timethreads notation. Timethreads are used to represent the typical causal flow patterns through a role team (intra-team timethreads) and across multiple teams (inter-team timethreads). Timethreads are composable meaning that a timethread can refer to another timethread as a timethread stub without showing the details of the stubbed timethread. This allows timethreads to be used for large scale causal flow patterns

(inter-team timethreads) that refer to smaller scale causal flow patterns (intra-team timethreads). Timethreads show the sequencing of activities, like methods activations and updates to role attributes. Timethreads also show the sequencing of architectural operations, e.g., start, stop, roll-in, roll-out, create, and destroy.

5. *Dynamic structure* is supported by operations that may be performed against components (roles, objects, and role teams). Create and destroy are simple low-level operations that define the end-points of a component's composite lifetime. These operations may be explicit or implicit in the architectural model. In general, a component is created when it is first required to operate in any place, and destroyed when it is no longer required in any place. A component *moves* from one place to another when it rolls into or out-of a placeholder. Timethreads are used as the sources of the architectural operations the effect dynamic structure, thus providing sequencing information that is not part of the role-based model.

The following issues were identified in Chapter 1 as needing further investigation.

... The use of causal-flow and timethreads to represent intra-group and inter-group behaviour patterns, the usefulness of representing a large portion of an existing framework in the role-based model, and the usefulness of representing inter-group dependencies and dynamic structure

These issues were addressed by applying the role-based model to the HotDraw object-oriented framework. The case study illustrates that architectural models and visual notations can be applied to object-oriented frameworks. It is the author's opinion that the model presented here provides a different perspective on system architecture than is offered by the current mainstream object-oriented modelling techniques. These techniques typically emphasis class-based design using E-R models and communicating state machines.

1. *The use of causal-flow and timethreads to represent intra-group and inter-group behaviour patterns*: The advantage of timethreads is that they provide a global perspective of behaviour that is lacking in current object-oriented models and technologies. This is especially important since object-oriented systems tend to exhibit a control structure that is distributed among many objects. Timethreads work best at a level where it is

appropriate to defer control-flow details. This makes inter-team timethreads perhaps more powerful than intra-team timethreads. At the level of individual teams, the structure of the team and the responsibilities of the roles may be enough, in some cases, for the timethreads to be inferred. Designers may want alternate more detailed representations, e.g. message sequence charts, at the level of individual teams.

2. *Representing a large portion of an existing framework in the role-based model* is useful. Typically, users that are new to a framework learn its design using a source code browser and reading manuals. It is difficult with only these perspectives to see the large-scale architecture of the framework because they present the system in fragmented pieces. Role teams support separation of concerns but may also be composed into larger structures with details suppressed. However, diagrams with many role teams can easily become complicated and multiple diagrams become necessary.

3. *Representing inter-group dependencies and dynamic structure* helps one to understand how a system operates through time. Other models represent similar issues but do so on an individual object basis. For example, state machine models of objects represent the roles that an object plays as states. The current state defines the set of messages that an object may accept at a point in time. If an object may accept multiple sets of messages (e.g., be in multiple roles) the notion of *and-states* [23] may be used. The role-based model is different because it expresses the operation of a group of objects when each object is in a specific state (one may view the role-based model as based on aggregate behaviours of multiple objects). It is believed that this view better supports the philosophy that *no object is an island* [7]. The role-based model presents a limited view of the structural dynamics that may occur in a running system. The dynamics that are represented are believed to be the important ones from an architectural design perspective.

6.2 Results

The results of this thesis are: a meta-model of software architecture; a precise definition of the temporal properties of the meta-model, including dynamic structure; a role-based model appropriate for object-oriented systems and frameworks; a visual design notation for the role-based model; and an application of the notation to an existing object-oriented framework.

The role-based model divides a system into a collection of small role teams. The model and notation for it are successful if the role teams are understandable in isolation and if the global architecture of the system can be understood when the teams are composed. The criteria for success is clear, but measuring the amount success is very difficult because of the number of variables that contribute to an overall success rating. Examples of some the variables include: ease of use, familiarity of the concepts to designers, suggestiveness of the notation, distance of the concepts from implementation technologies (i.e., ease of mapping the concepts to languages and environments), availability of methods and heuristics, automated support, clearness of the documentation supporting the model, examples of use, background of designers, and the characteristics of the systems being modelled.

It is the author's opinion that the role-based model meets the criteria for success. A more subjective evaluation could be obtained by conducting experimental trials that measure the performance of designers that create or use the role-based model, and with enough resources, the role-based could be evaluated against existing models. However, the scientific validity of such trials is suspect given the number of uncontrolled variables. The best approach, although time consuming and imprecise from a pure scientific standpoint, is to apply the model to large a number of case studies, preferably performed by someone other than the models originator, and to report informally on the findings (Karam and Casselman refer to the maturity of a software method in these terms [39]). Doing so is outside the scope of this thesis.

6.3 Future Work

6.3.1 Case Studies

More case studies are needed that use the role-based model. Reverse-engineering existing systems (an under-used approach) reveals the abstractions that designers use and also the problems that result when the proper abstractions are not used. Forward-engineering to create new systems demonstrates the usefulness of the design concepts as thinking tools. Both approaches need to be pursued and the author expects, given the immaturity of the role-based model, that doing so may indicate where changes are needed to the model and its notation.

6.3.2 Animation

Although the role-based model expresses elements of behaviour and dynamic structure, the diagrams produced are static 2 dimensional images. From these diagrams designers must infer the actual runtime structure of a system as it changes through time. It may be possible to adapt the notation of the role-based model for use in tools which trace the actual runtime architecture of executing object-oriented programs. For example, objects could be represented and collaboration groups (role teams) could be overlaid on them. Wires between objects could display message history information, and execution trails could be abstracted into timethreads if enough information on the expected timethread patterns are given to the tool. Layering could be used to organize the collaboration groups in a hierarchy to reflect the abstract nature of the groups. The DOORS project [12] in the Department of Systems and Computer Engineering at Carleton University is currently researching the use of such an approach in conjunction with 3 dimensional imaging and powerful techniques for traversing the models.

6.3.3 Design Methods

Design methods are needed to support forward-engineering of systems using the role-based model. There are two approaches: the first integrates the role-based model with an existing design method, and the second develops a new method.

The first approach is profitable because it builds on existing work, but may be difficult if the existing method is not easily adapted. This approach also requires that the concepts of the role-based model be analysed in terms of the concepts of the existing method. Of particular importance are the areas where the models overlap because it is at these points that the models must be kept consistent. It is the author's opinion that some mix of Responsibility-Driven Design [7], Rumbaugh's Object Modelling Technique [44], Timethread Driven Design [14], and the Role-based Model would make a powerful combination. The CRC cards of Responsibility-Driven Design are powerful in their flexibility during the early stages of design. Timethreads are a good mechanism for recording the results of a CRC card session. Rumbaugh's Object Model Diagrams are powerful because they can present a class-based view of large portions of a system. The role-based model is a complimentary

view to Rumbaugh's Object Model Diagrams. However, links between the two models are needed. Timethread-driven design can be used to develop a role-based model.

A new method would use the role-based model as the governing philosophy of the method [39], meaning it would be the central model on which the design method would revolve. In such a method, roles would take centre stage over objects and classes, and role teams (i.e., important system objectives) would be identified early in the design process. Causality-flow would be used to identify collaborations between roles, identify role teams, assign responsibilities to roles, and design interaction patterns between roles. Roles would be aggregated together to identify classes.

In either case, heuristics and guidelines are needed to support role-based modelling. Important questions that are not explicitly addressed in this thesis include: (1) What criteria are used to identify a role team, i.e., what constitutes an important system objective? Maintaining an invariant is one criteria but there must be others. (2) What are the guidelines for implementing a role-based model in standard object-oriented programming languages? For example, what are some standard and safe approaches for implementing the propagation of destroy operations across equivalence relationships? (3) What are the criteria for using fixed roles, optional roles, and role placeholders? For example, a fixed role in an optional role team can be operationally equivalent to an optional role in a fixed role team. (4) What constitutes an important causal-flow path? And a related question: What is the appropriate level of abstraction to use when expressing causal-flow patterns? Being to abstract, may create too great a distance to the implementation; being too detailed, may result in complex diagrams, (5) Are there standard architectural patterns that may be used as templates for organizing a role-based model? ---for organizing causal-flow patterns? (6) What constitutes a good role-based model?

6.3.4 A Meta-Model for Structural Dynamics of Software Systems

The meta-model presented in this thesis is rather narrowly focused to simplify the presentation but could generalized. For example, placeholders for methods could be added. This would allow the behaviour of a component to be changed dynamically at runtime by rolling different methods into and out-of the method placeholder. This would be similar to the con-

cept of code blocks in Smalltalk [22]. The placeholder concept could be generalized to include *designed-in* internals. A component filling a placeholder would be required to provide plug compatible internals to an arbitrary nesting level. Wires are currently fixed but moveable wires (i.e., wires that may be re-routed) may be a more general concept (Hermes [47] supports this notion).

Generalizing the meta-model would require an extensive study of existing architectural models of software systems and many case studies to evaluate the usefulness and compatibility of the more general concepts.

6.3.5 Formalisms and Languages

It seems to be natural for formalisms and programming languages to be driven by the concepts that designers use to build systems. Given that the role-based model is useful, one would expect that formalisms and languages will be developed to support it. Evidence already exists: Helm [26] has developed a formalism for collaboration groups, called contracts, and is developing tools for implementing systems in terms of contracts. Arapis has developed a formalism for collaboration groups, called contexts [3], and has developed techniques for proving that specifications in the formalism are self consistent. Helm also reports that a programming style that makes contracts (read role team) explicit in the implementation greatly aids system comprehension and maintenance.

There are several questions that must be addressed before the development of formalisms and languages.

What place do formalisms have in role-based modelling? The strength of the role-based model comes from its use of common sense concepts and the deferral of details. Any formalisms applied to the role-based model must not deter from this strength. A possible approach is to *hide* formalisms from the user behind the interfaces of intelligent tools. The tools would interpret user input, maintain a formal representation of the model, and answer user queries against the model. For example, tools could be used to determine if a more detailed model (i.e., more detailed architecture or timethreads) is still consistent relative to a more general one.

What aspects of the role-based model should be formalized? Formalizing the temporal properties of role playing would allow tools to enumerate the runtime architectures that are possible. Formalisms could be used to enter constraints that would further limit the runtime architectures. A formalism for timethreads and for the relationship between timethreads and the role-based model would allow for executable designs. Current research is addressing how the properties of objects and roles may be defined and matched to allow for object role playing [2].

What features are needed in a programming language to support role-based modelling? Explicit support for role teams is desirable. Role teams should be components of a language and should be identifiable runtime components. There should be explicit support for the architectural operations on components (start, stop, roll-in, roll-out, etc.). An interesting project would be to prototype an environment for architectural modelling by building an object-oriented framework based on the concepts of the meta-model. The relationships between the concepts of the meta-model would be a basis for organizing the classes of the framework. Such a prototype would be much easier to build than a language or tool and would enable one to easily experiment with new modelling concepts.

6.3.6 Tools for Role-Based Modelling

There are two questions to ask: What tools are needed for role-based modelling? And are there existing tools that are useful?

Answering the second question first, the most powerful tool for role based modelling that currently exists is the ObjecTime tool [45]. Many of the concepts of the role-based model are supported by ObjecTime. The differences are the following: the detailed semantics of fixed and optional components and the rules for equivalence are less general in ObjecTime; the notation for the role-based model is more powerful and successive than ObjecTime's notation; and behaviour is modelled using state machines in ObjecTime and using a combinations of timethreads, methods, and architectural operations in the role-based model. These differences aside, it is possible to created executable models in ObjecTime that can be interpreted in terms of roles and role teams.

A set of tools rather than a single tool is needed for role-based modelling. Minimally tools are needed which support drawing and maintaining the diagrams with timethreads overlaid on them. More powerful tools would support *execution* of models given the timethread specifications over the architectural models. The ability to organize role teams into reuse hierarchies would support the develop of frameworks for architectures. It should be possible to retain the timethread specifications and use them to verify the operation of a system in an automated way.

Source code is not likely to be completely eliminated from the designer even with more powerful design tools. Therefore, source code browsers should have access to the design models such that the source may be presented and organized in terms of the design models. Execution tracing and debugging tools should be able to present the architecture of a running object-oriented system in role and timethread terms. The existing software-base will continue to be a reality. Therefore, tools are needed to support reverse-engineering of existing systems into the role-based model and timethreads. Completely automatic approaches are not likely to be as powerful as tools which acts as assistants to a designer who is in charge of the reverse-engineering effort.

References

[1] G. Adams, *Describing Groups of Interacting Objects Using Path Expressions,* Master's thesis, Dept. of Computer Science, Carleton University, Ottawa, Ontario, CAN., May 1992.

[2] E. P. Anderson and T. Reenskaug, "System Design by Composing Structures of Interacting Objects," In *ECOOP'92 European Conference on Object-Oriented Programming,* pp. 133–152, Springer-Verlag, Utrecht, The Netherlands, June/July 1992.

[3] C. Arapis, "Temporal Specifications of Object Behaviour," In G. Goos and J. Hartmanis, editors, *MFDBS 91 3rd Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems,* pp. 308–324, Springer-Verlag, Rostock, Germany, May 1991.

[4] S. Bailin, "An Object-Oriented requirements Specification Method,"*Communications of the ACM*, Vol. 32, No. 5, pp. 608–623, May 1989.

[5] M. Baldassari, G. Bruno, and A. Castella, "PROTOB: an Object-Oriented CASE Tool for Modelling and Prototyping Distributed Systems," *Software Practice and Experience*, Vol. 21, No. 8, 1991.

[6] K. Beck, "Seminar on Responsibility-Driven Design," IBM Mentored Object-Oriented Design Series, Thornwood, NY, USA, May 1993. *(unpublished)*

[7] K. Beck and W. Cunningham, "A Laboratory for Teaching Object-Oriented Thinking," In *Proceedings of OOPSLA'89*, pp. 1–6, ACM/SIGPLAN, New Orleans, Louisiana, October 1-6, 1989.

[8] G. Booch, *Object-Oriented Design*. Redwood City, California:Benjamin/Cummings, 1991.

[9] G. Booch, "Object-Oriented Development," *IEEE Transactions on Software Engineering*, Vol. 12, No. 2, February 1986.

[10] R.J.A. Buhr, *Pictures that Play for Designing Concurrent Real-Time Systems,* TR-SCE-91-08, Ottawa, CAN., April 1991. (*to appear Software Practice and Experience*)

[11] R.J.A. Buhr, *Practical Visual Techniques in System Design (with Applications to Ada)*. Englewood Cliffs, NJ:Prentice-Hall, 1990.

[12] R.J.A Buhr, *Second Progress Report CRD Grant #661-008/90 Object-Oriented Design of Real-time Systems: The Doors Project,* Dept. of Systems and Computer Engineering, Carleton University, Ottawa, CAN., Sept 10, 1992.

[13] R.J.A. Buhr and R.S. Casselman, "Architectures with Pictures," In *Proceedings of OOPSLA'92*, pp. 466–483, ACM/SIGPLAN, Vancouver, Canada, October 1992.

[14] R.J.A. Buhr, R.S. Casselman, and B. Foster, *Designing with Timethreads*, TR-SCE-93-05, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, CAN., *(In progress)*.

[15] R.J.A. Buhr, R.S. Casselman, M. Petras, and J. Anderson, *Object-Orientation and the De-*

sign of Large Real-time Systems, TR-SCE-91-38, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, CAN., November 1991.

[16] P.P.S. Chen, "The Entity Relationship Model - Toward a Unified View Of Data," *ACM Transactions on Database System*, Vol. 1, No. 1, March 1976.

[17] P. Coad and E. Yourdan, *Object-oriented Analysis*, Yourdan Press, Prentice Hall, 1990.

[18] D. Coleman, F. Hayes, and S. Bear, "Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design," *IEEE Transactions on Software Engineering*, pp. 9-18, Vol. 18, No. 1, January 1992.

[19] L. P. Deutsch, "Design Reuse and Frameworks in the Smalltalk-80 System," In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability*, chapter 3, pp. 57–71, ACM Press, 1989.

[20] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*. Addison Wesley, 1990.

[21] D. Garlan and M. Shaw, "Advances in Software Engineering and Knowledge Engineering," Volume 1, World Scientific Publishing Company, 1993.

[22] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementatio*n. Reading, MASS:Addison-Wesley, 1985.

[23] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, Vol. 8, 1987.

[24] D. Harel, "Biting the Silver Bullet: Toward a Brighter Future for System Development," *Computer*, pp. 8-19, VOL. 25, NO. 1, January 1992.

[25] F. Hayes and D. Coleman, "Coherent Models for Object-Oriented Analysis," In *Proceedings of OOPSLA'91*, pp. 171–183, ACM/SIGPLAN, Pheonix, Arizona, October 1991.

[26] R. Helm, I. Holland, and D. Gangopadhyay, "Contracts: Specifying Behavioural Compositions in Object-Oriented Systems," In *Proceedings of OOPSLA/ECOOP'90*, pp. 169–180, ACM/SIGPLAN, Ottawa, CAN., October 1990.

[27] J. Hogg, "Islands: Aliasing Protection in Object-Oriented Languages", In *Proceedings of OOPSLA'91*, pp. 271–285, ACM/SIGPLAN, Pheonix, Arizona, October 1991.

[28] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt, "The Geneva Convention On The Treatment of Object Aliasing", OOPS Messenger, VOL. 3, NO. 2, pp. 11–16, ACM/SIGPLAN, April 1992.

[29] I. M. Holland, "Specifying Reusable Components in Contracts," In *Proceedings of ECOOP'92 European Conference on Object-Oriented Programming*, pp. 133–152, Springer-Verlag, Utrecht, The Netherlands, June/July 1992.

[30] ISO, *LOTOS --- A formal Description Technique Based on Temporal Ordering of Observable Behaviour* , International Organization for Standardization, ISO 8807, 1988.

[31] I. Jacobson, M. Christeron and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley, ACM Press, 1992.

[32] I. Jacobson, "Object-Oriented Development in an Industrial Environment," In *Proceedings of OOPSLA'87*, pp. 183–191, ACM/SIGPLAN, Orlando, Florida, October 1987.

[33] I. Jacobson, "Reengineering of Old Systems to an Object-oriented Architecture", In *Proceedings of OOPSLA'91*, pp. 340-351, ACM/SIGPLAN, Pheonix, Arizona, October 1991.

[34] M. Jackson, *System Development*. Englewood Cliffs, NJ:Prentice-Hall, 1983.

[35] R. E. Johnson, "Documenting Frameworks using Patterns," In *Proceedings of OOPSLA'92*, ACM/SIGPLAN, Vancouver, B.C., CAN., October 1992.

[36] R. E. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-oriented Programming (JOOP)*, Vol. 1, No. 2, pp. 22-35, June/July 1988.

[37] R. E. Johnson and V. Russo, *Reusing Object-Oriented Designs*, TR-UIUCDCS-91-1696, Department of Computer Science, University of Illinois, Urbana-Champaign, May 1991.

[38] C. Jones, *Systematic Development Using the VDM Approach*. Prentice-Hall Intl., 1986.

[39] G.M. Karam and R.S. Casselman, "A Cataloging Framework for Software Development Methods," *Computer*, Vol. 26, No. 2, pp. 34–47, February 1993.

[40] G. Kiczales, J. des Rivieres, and D.G. Bobrow, *The Art of the Meta-Object Protocol*, MIT Press, 1991.

[41] G. E. Krasner and S. T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming (JOOP)*, Vol. 1, No. 3, pp. 26–49, September 1988.

[42] G. Menga, G. Elia, and M. Mancin, *G++: An Environment for Object-Oriented Design and Prototyping of Manufacturing Systems*, Corso Duca degli Abruzzi 24, 10129 Torino, Italy, 1991.

[43] J.L. Peterson, "Petri Nets," *ACM Computing Surveys*, Vol. 9, No. 3, pp. 223–252, September 1977.

[44] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ:Prentice Hall, 1991.

[45] B. Selic, G. Gullekson, J. McGee, and I. Engelberg, *ROOM: An Object-Oriented Methodology for Developing Real-Time Systems*, In *Proceedings of CASE'92*, pp. 230-241, Montreal, Canada, July 6-10, 1992.

[46] S. Shlaer and S.J. Mellor, *Object Lifecycles Modeling the World in States. Yourdon Press Computing Series*, Englewood Cliffs, NJ:Prentice Hall, 1992.

[47] R. E. Strom and et. al., *Hermes a Language for Distributed Computing*. Englewood Cliffs, New Jersey:Prentice Hall Series in Innovative Technology, 1991.

[48] S. Subramanian, *Controlling the Structural Evolution of Software Systems*, PhD thesis, Computer Science and Engineering: University of Michigan, 1988.

[49] M. Vigder, *Applying Formal Techniques to the Design of Concurrent Systems*, PhD thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, CAN., June 1992.

[50] C.A. Vissers, G Scollo, M. van Sinderen, and E. Brinksma, "Specification Styles in Distributed Systems Design and Verification," *Theorectical Computer Science*, pp. 179-206.

[51] P. T. Ward, "The Transformation Schema: An Extension to Data Flow Diagrams to Represent Control and Timing," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, pp. 198–210, February, 1986.

[52] R. Wirfs-Brock and R. Johnson, "Surveying Current Research in Object-Oriented Design," *Communications of the ACM*, Vol. 33, No. 9, pp. 104–141, 1990.

[53] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*. Englewood Cliffs, NJ:Prentice Hall, 1990.

[54] U.S. Department of Defence, *Reference Manual for the Ada Programming Language*, MI-STD-1815a, 1983.

[55] D. de Champeaux, D. Lea, and P. Faure, *Object-Oriented System Development*. Addison-Wesley, 1993.

Appendix A: Timethread Notation

A ***timethread*** is a cause-effect path through a system linking activities and components. A timethread starts at some point of stimulus, touches the elements of a design in the order they are activated, and ends at a point that marks the end of further processing or the delivery of some response [13]. Time increases monotonically along the path of a timethread.

Visually a timethread is drawn as a curved line that connects elements of a design. Figure 76 gives an example of a timethread overlaid on a generic wiring diagram. The starting point of the thread is a filled circle and the ending point is a straight line. The starting point is drawn at the place in the system that generates the stimulus that creates the timethread instance. A starting point that is drawn outside of the system indicates that the starting stimulus comes from the environment. The ending point is drawn at the place where the thread terminates or delivers its final response.

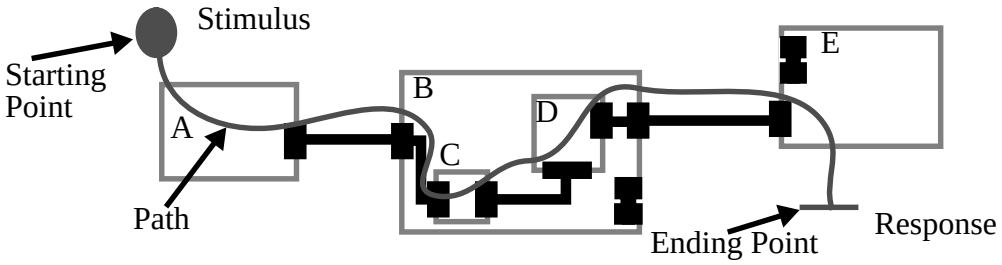

Figure 76: Behavioural Fundamentals: Timethreads

The timethread in Figure 76 specifies that the places are activated in the following order in response to a stimulus from the environment: **A**, **B**, **C**, **D**, and **E**. After **E** is activated a response is delivered to the environment. The nature of the activities performed is not expressed by the timethread itself. Supporting information in the form of annotations to the timethread or more detailed documentation about the components involved is required.

Timethreads are a higher-level behavioural concept than control-flow. In fact, in a concurrent system a timethread may span several threads of control. The purpose of a timethread is to show temporal sequencing; timethreads do *not* show which elements of a design are responsible for achieving the sequencing. Thus, timethreads omit details like who calls

whom and how control is returned between design elements. Section 2.6.1 discusses the relationship between timethreads and other behaviour specification techniques that are common in the literature on object-oriented design.

Timethreads have proven useful for specifying and designing concurrent systems [14]. In this thesis, a subset of the timethread notation that is applicable to sequential systems is used as illustrated by Figure 77.

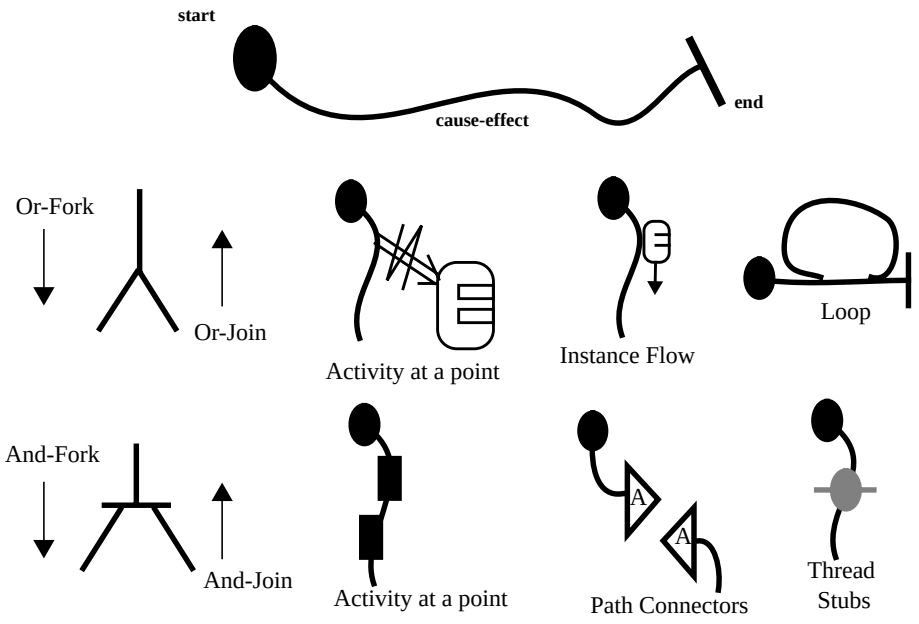

Figure 77: Subset of the Timethreads Notation

An ***or-fork*** is a split of a timethread path into alternate paths. One of the alternate paths is taken based on some condition in the system. The condition may be specified as a textual annotation. An ***or-join*** is a point on a timethread path where alternate paths merge to a common path. An ***and-fork*** splits a timethread into timethread paths, each timethread path will eventually be taken, but the order in which the paths are taken is random (annotations could be added to make the order explicit). In general, the paths leaving an and-fork could proceed in parallel, but for sequential systems some arbitrary order must be chosen. Multiple timethread paths may rejoin at ***and-join***. The timethread path leaving an and-join proceeds

once all the incoming timethread paths have arrived.

Annotations may be used along a timethread path to indicate the place and time at which at some activity occurs, see ***activity at a point*** in Figure 77. It may be useful to use symbols from the structural design notation to annotate points along the timethread. For example, the double lined arc in the figure with a lightening bolt through it could represent the destruction of a component at a point in time. Other possible uses for this style of annotation may include starting a component, stopping a component, or creating a component. Solid filled boxes represent activities that are performed in the sequence specified by the timethread. This is a useful annotation when the activities are known but have not yet been allocated to components.

One may show the flow of component instances and primitive data elements along a timethread by putting ***instance flow*** icons next to the thread. The shape of the instance may be changed to the shape of the architectural entity that it represents. ***Path connectors*** are a diagramming convenience for routing a timethread path between two points on a diagram. No activity occurs between the path connectors. A timethread stub represents hidden detail at a point along the path of a timethread. Timethread stubs allow for nesting in a thread specification and may also be used to show sequencing of complex activities. A ***loop*** allows for a common portion of a timethread to be repeated. Annotations may be used to specify the looping conditions.

Appendix B: Rumbaugh's E-R Diagramming Notation

The entities of an E-R model may be identifiable things, such as computer parts, people, and public institutions; or the entities may be abstract concepts, such as colour, beauty, and quality. The relationships of an E-R Model describe how the entities are related, for example: a keyboard is part-of a computer (an *aggregation* relationship); red is a kind of colour (an *is-a* relationship); and people request services from public institutions (a *general association*). E-R modelling is very useful for organizing and presenting ideas at a high level abstraction and is used for this purpose by many object-oriented design methods.

Rumbaugh [44] presents an E-R model for object-oriented development that he calls an Object Model. Figure 78 presents a portion of an Object Model that describes the structure of written documents. Such a model could be used for the production of a word processing program. The entities of an Object Model are classes and the arcs are relationships between classes or instances of classes (objects). An *aggregation* relationship is drawn with a diamond on the end of a relationship arc next to the class that acts as the container, e.g., a **Paragraph** is part-of a **Document**, a **Word** is part-of a **Paragraph**, and a **Character** is part-of a **Word**. A filled circle on a relationship arc next to a class specifies that zero or more of that classes instances (objects) may participate in the relationship, e.g., a **Document** consists of many **Paragraphs** and many **Pagelayouts**. The number of objects in a relationship may be constrained by a textual annotation, e.g., a **Word** consists of one or more **Characters** as specified by the 1+ annotation next to the **Character** class. When there is no annotation specifying the number of objects in a relationship, the default number is exactly one. An *is-a* relationship is annotated with a triangle; more general entities are connected to the apex of the triangle, and more specialized entities are connected to the base of the triangle. For example, **Letters**, **Forms**, and **Theses** are specialized types of **Documents**; a **Theses** may be further categorized as a **Masters** or **Ph.D.** theses. A *general association* between classes is drawn as a simple arc. By its nature a general association is bi-directional, although the name of the association usually reads in only one direction; e.g., a **Character** is *wriiten in* a **Font**. To make the reading of associations clear, Rumbaugh's notation is extended here to have arrowheads on arcs. The direction of the arrow on an association arc does not imply any details like control direction or pointer access.[1]

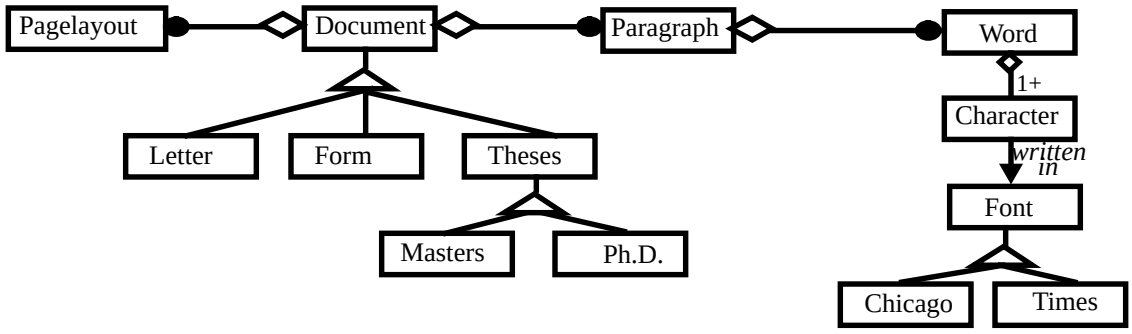

Figure 78: Example E-R Model

Figure 79 uses a few small examples to explain other features of Rumbaugh's notation. A class may have *attributes* and *operations* that describe an objects responsibilities in a model; they appear as text strings inside the class box beneath the name of the class. An attribute is a data value held by an instance of a class, e.g., a **Filing Folder** may have a creation date. An operation describes some action performed by instances of the class, e.g., a **Filing Folder** may provide operations to open, lock, and add files to itself. A *ternary association* joins three classes in a common relationship, e.g., a **Programmer** uses a particular **Programming Language** on a particular **Project**. *Recursive aggregation* results when an object consists of entities of its own type or objects that are derived from it through an is-a relationship. An example of recursive aggregation may involve a **Filing Folder** which is composed of other **Filing Folders** internal to it. Relationships between objects may have attributes and operations of their own. For example, the *works for* relationship between **Person** and a **Company** may have attributes that specify the **Person's** salary and job title, and an operation to calculate an employees allowable leave. Using relationship attributes is cleaner than assigning the attributes to the person object, because a person may have many salaries and job titles if he works for many different companies.

[1.] Rumbaugh uses an undirected arc for relationships and relies on context to make the direction of the relationship understandable.

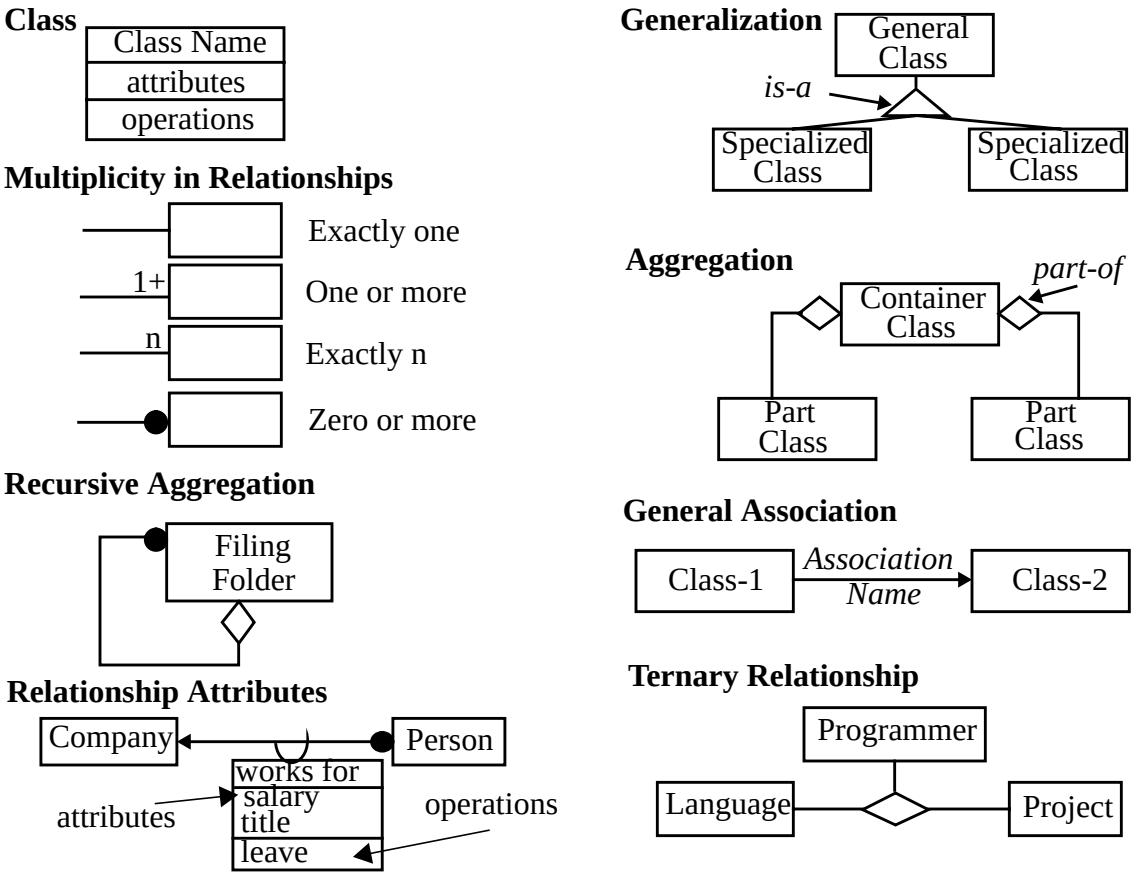

Figure 79: Subset of Rumbaugh's Object Model Notation

Appendix C: Refinements to the Meta-Model

This appendix presents some further details of the meta-model that were not presented in the body of thesis.

Notational Issue: Replication

Replication in an architectural design diagram is used to represent the possibility that more than one architectural entity will be present when a system is in operation. This is an important concept because it allows a design diagram to be a template for many possible runtime configurations. Replication also provides a means of specifying the arity of architectural entities in their relationships with one another. For example, a simulation of a classroom may model the teacher and the students as architectural entities. The interaction pattern between the teacher and each student would be identical and could be specified in a generic way by modelling the students as replicated. Replication alone does not imply dynamic structure; it is a simply a convenient way to represent repeated architectural forms.

Figure 80 provides an example of a replicated place (**A**) and a replicated terminal (**B**). The architectural design diagram uses shading to show replication and the number of replicated entities may be constrained using a textual annotation. The right-side of the figure shows some possible runtime configurations for the architectural design diagrams on the left. Replication does not imply that identical copies of the replicated architectural entity occur at runtime. For example, **A1**, **A2**, and **A3** in Figure 80 are not necessarily identical copies of one another. The only restriction is that the interface properties of components **A1**, **A2**, and **A3** are compatible with the specification of **A** (again, compatibility is deferred at this level). In the case of the replicated terminals, **B1** and **B2** must be compatible with the wiring protocol of **B**.

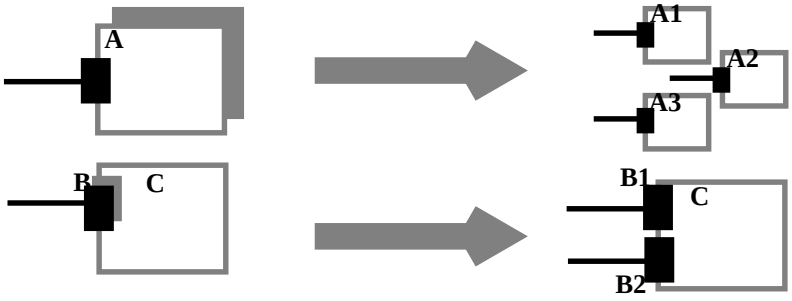

Figure 80: Replication in Architectural Design Diagrams

Detailed Wiring Issues: Dynamic Routing

In the architectural meta-model, *dynamic routing* in wiring diagrams is the ability to route a message across the appropriate wire based on the contents of the message. In Figure 81, a terminal is imagined to have an internal agency for this purpose (note that this figure is used to explain the properties of the meta-model and does not introduce any new notation). The dynamic routing property of terminals allows a single terminal to be bound to multiple wires, as shown in Figure 81.

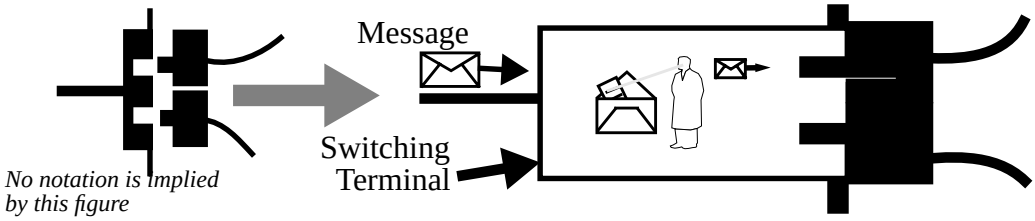

Figure 81: Terminals with Multiple Wires direct Messages over the Proper Wire

Dynamic binding, as defined in relation to object-oriented programming langauages, is subtely different from dynamic routing. *Dynamic binding* in programming languages is the ability to route messages to the appropriate destination (i.e., method) at runtime based on the class (type) of the object receiving the message. Polymorphism through inheritance is achieved at runtime by dynamic binding: the actual target method invoked is determined at runtime by traversing the inheritance hierarchies. The differences between the concepts are: (1) dynamic binding is defined in terms of classes (types) whereas dynamic routing is

defined in terms of component instances that are bound by wires, and (2) the dynamic binding that occurs due to inheritance in object-oriented languages is not considered part of dynamic routing, because the inheritance trees are collapsed in the architectural view of systems, see Section 2.1.5. Polymorphism, the ability to send messages to objects of different classes (types), is an inherent part of the models presented in this thesis, because any component may be participate in a place in a wiring, regardless of how it is constructed in terms of types, provided its has the necessary properties to participate in the place.

The combination of replication and dynamic routing at terminals has an effect on a system's runtime configuration, i.e., a system's wiring at runtime. Figure 82 shows an example of an architectural diagram that includes a replicated place and a replicated terminal. The right-side of the figure shows one possible runtime configuration in which there are two **A** places (**A1** and **A2**) and two **B** terminals (**B1** and **B2**). The replication of wires in the runtime configuration is implied by the replication of places and terminals in the architectural diagram, e.g., there are multiple wires connected to the terminals of **D**. It is a terminal's ability to route messages that makes this runtime configuration possible because without it a terminal may be connected to only one outgoing wire. [1]

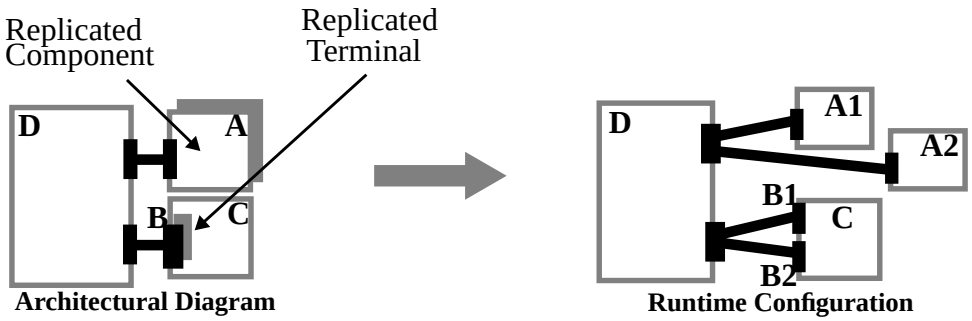

Figure 82: Replication of Components and Terminals implies Replication of Wires

A ***detached terminal*** is not connected to the interface of place, but is part of the internals of some place. Figure 83, illustrates how the detached terminal concept may be derived

[1] Early versions of the ObjecTime [45] tool do not support routing of messages at terminals. Replication is handling by duplicating terminals such that a terminal is only bound to one wire. Operationally the effect is the same as here, but may complicate somewhat the management of identifiers because message routing must be performed manually. Newer versions of the tool have message routing at terminals.

from the meta-model (top of figure) and its notation (bottom right of figure). Detached terminals are useful for creating junction points in wiring diagrams that are removed from the interfaces of places. Specific derivates of the detached terminal concept could act as simple multiplexers or protocol adapters (not developed further here). ***Routers*** are one type of detached terminal that are drawn as filled boxes at places where wires split. Like all terminals, a router is able to route messages that arrive to it to an appropriate outgoing wire, and it does not take any actions other than routing. Routers are useful when a message path among components shares a common wire. When a wire is shared there are two different interpretations possible. In Figure 83(a) a shared wire is split into several branches with no annotation at the point where the wires split. In this case, a message on the shared wire will be sent along all the branches; therefore, a split wire is like a single wire with multiple connection points. In Figure 83(b) a router is drawn as a small filled box at the point where the wires split. The interpretation in this case is that a message travelling over the common wire is sent by the router to one of the outgoing wires. When used as in Figure 83(b), routers can help reduce the number of wires in a wiring diagram.

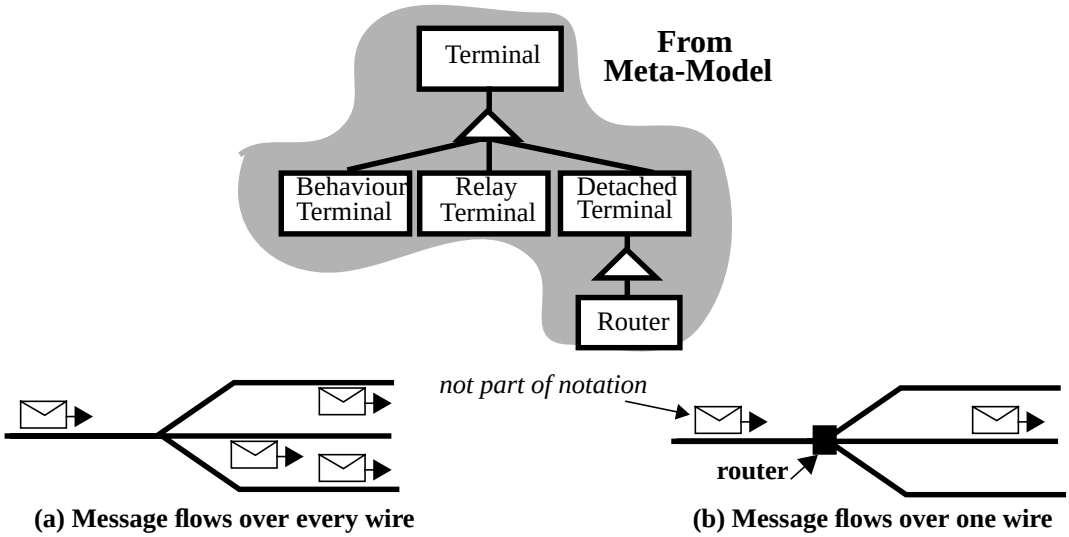

Figure 83: Routers for Message Routing

Cables and Layers

Often times a group of wires are so closely related that it is convenient to the think of the group as a logical entity; e.g, the black, white and ground wires in household wiring. Other

times wiring diagrams can become so complex with a multitude of wires that some form of abstraction is needed to reduce the clutter. For these cases, the meta-model may provide ***cables***, ***cable connectors***, and ***cable terminals*** as shown in Figure 84. A cable is a bundle of wires that may be treated as a single entity; it is drawn as a three parallel lines. The wires in a cable may be terminated as a group by a ***cable connector***; a ***cable connector*** is composed of many connectors. A ***cable terminal*** is composed of many terminals and may be connected to a cable with a compatible cable connector.

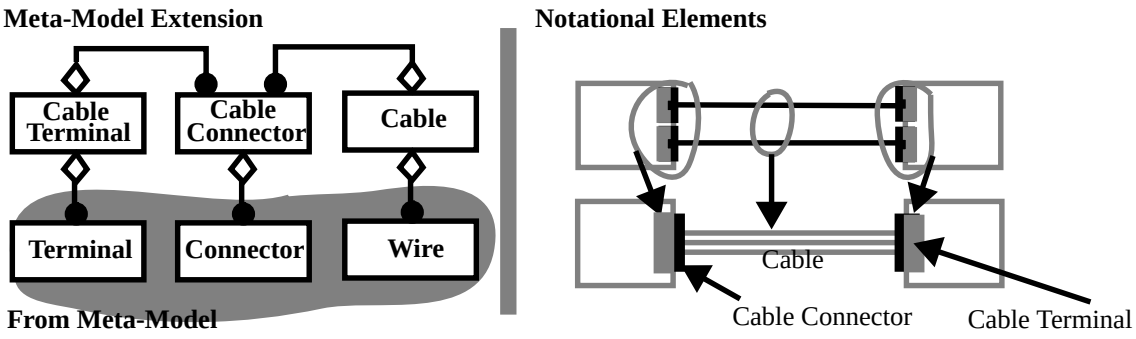

Figure 84: Cables and Cable Connectors

Another way of having too many wires occurs when one component is used by nearly every other component in a wiring. Then a factoring of the wiring into layers may be appropriate. Figure 85 models a layer as a specialization of the component concept: a layer may have many internal components, and a layer may be ***used by*** many other layers. The ***used by*** relationship is a many-to-many relationship (note the black dots on both ends of the relationship arc); therefore, a layer may be used by many other layers and a layer may in turn use many other layers. The layer concept supports the factoring of a system into a hierarchical stack of layers with the lower layers providing service abstractions to the higher layers.

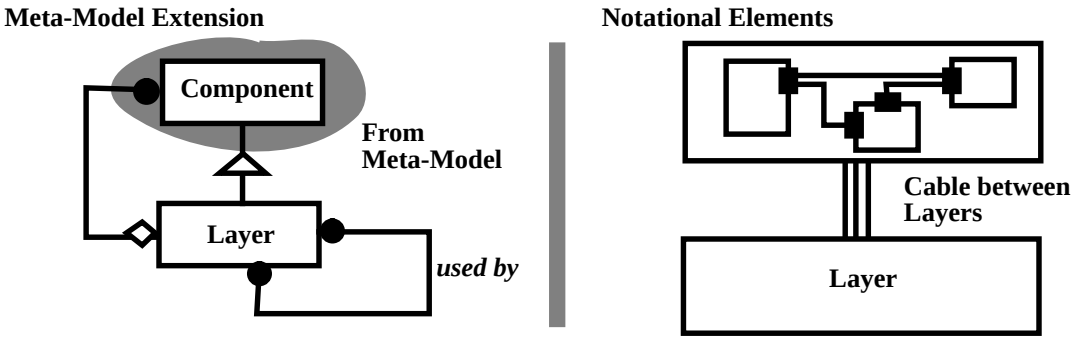

Figure 85: Factoring Wiring Diagrams into Layers

Notationally a layer is represented as a component that is connected to another layer by a cable. Terminals and connectors have been dropped because, for design purposes, the abstractions provided by a lower layer are treated as givens by the layer above, and adding explicit wires would obscure the layering concept. The factoring into layers may introduce the need to add new wires and components in a higher layer to include the abstractions that are provide by the lower layer. This need has inspired whole design notations, e.g. MachineCharts [11].

The cable between layers may be bidirectional, if the layers communicate back and forth, or unidirectional, if one layer acts strictly as a client and the other as a server. In the latter case, a unidirectional cable may be drawn between the layers; it is drawn as a cable with an arrowhead from the client to the server.

Operations on a Replicated Group of Components

When replication is used one must distinguish between operations that apply to the group as whole versus those that apply to individuals of the group. These cases are distinguished in the following way: operations that apply to the group are directed at the shadow of the replicated structural element, and operations the apply to individual components of the group are directed at the structural element that specifies the properties of the group.The example of Figure 86 shows how to start one optional component in a replicated group, and how to stop all the optional components in a replicated group.

Figure 86: Operations on Replicated Components

Replication and Dynamic Structure

When using replication, the possible runtime configurations may be constrained by annotating the replicated element with text. For example, a 3 next to a replicated component would say that the same component specification is repeated three times in the wiring. In this case, replication is simply a shorthand for drawing a repeated-static structural form. Without a constraining annotation, the number of replicated elements is left open-ended, and replication amounts to a template for creating structural organizations at runtime. For example, Figure 87 shows a unconstrained replicated optional component and an unconstrained replicated placeholder. Each time the replicated optional component **C** is started a new component instance of **C** appears in the wiring. Whenever the replicated optional component **C** is stopped, one of the instances of **C** is stopped and removed from the wiring. For the replicated placeholder, each roll-in would require a new placeholder in the wiring if there are no empty ones available, and each roll-out operation would free up one of the placeholders. The assumption is that there is some mechanism outside of the meta-model for managing the identifiers of component and placeholder instances in the above cases.

Fixed components may also be replicated. The replication factor of a fixed component must be known at the time when the container of a fixed component becomes operational. A statically defined replication factor could be set before the system begins operation, or the replication factor could be set during an initialization phase of a container before it begins its operational lifetime. In design diagrams that leave the replication factor unspecified, the assumption is that the replication factor will become known during the initialization phase of the container of the fixed component.

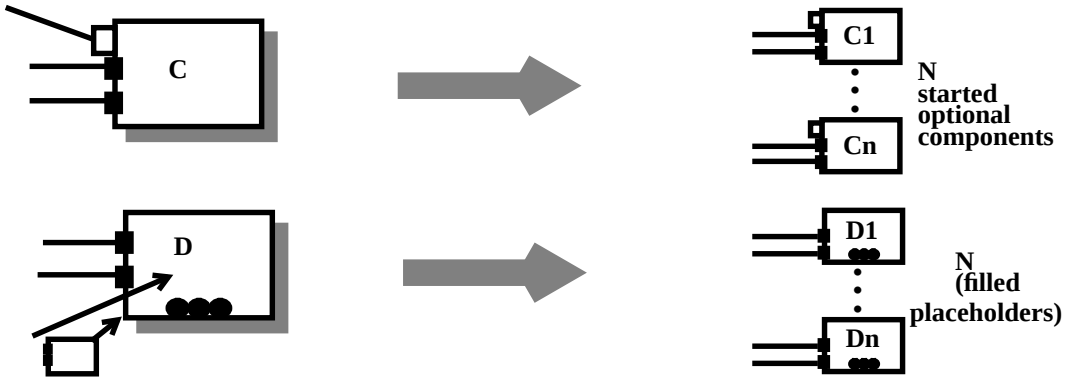

Figure 87: Replication as a Template for Runtime Structural Forms

Multiple Part-of and Replication

Figure 88 defines some rules that govern the use of explicit replication factors when used with equivalence. Recall that an explicit replication factor is used to define the precise number of structural elements that will occur at runtime. When a replication factor is not given, the number of structural elements that may occur at runtime is arbitrary; however, that number may be constrained through equivalence. When replication factors are explicit, the components joined in an equivalence must have matching replication factors, see case **I** of Figure 88. If any one of the replication factors is specified in an equivalence among components, then the replication factor in all places joined by the equivalence is fixed to that factor. For example, if **#A** were specified in case **I** but not **#B**, then the replication factor across the equivalence would be fixed to **#A**. Compound replicated places are allowed as in case **II**. In this case, the total number of replicated components across the equivalence must match when the multiplication factor resulting from compound replication is considered. Case **III** illustrates that the replication factor of placeholders need not match across an equivalence. This is allowed because of the flexibility that placeholders provide in regards to how they are filled with component instances.

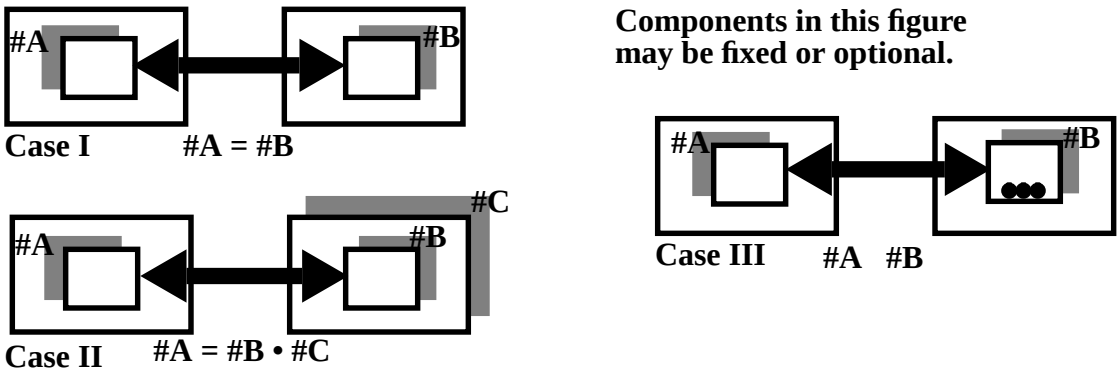

Figure 88: Explicit Replication Factors and Multiple Part-of

Replication factors that are unspecified are determined dynamically, but the total number of components may be constrained through an equivalence relationship. Figure 89 illustrates the case of a fixed component equivalenced to an optional one. When container **A** comes into operation the number of components (**B**) in **A** may be fixed during the initialization phase of **A**. Before **A** begins, however, **C** may start an arbitrary number of components (**D**). The number started by **C** must be less than or equal to the number that **A** requires, otherwise, there would be a runtime error. If **A** began before **C**, than **C** could subsequently start as many components as were required by **A**. If **C** started more than this number of components, there would be an error.

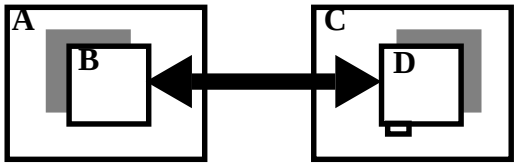

Figure 89: Deferred Replication Factors and Multiple Part-of