A Survey of Structured and Object-Oriented Software Specification Methods and Techniques

ROEL WIERINGA

University of Twente

This article surveys techniques used in structured and object-oriented software specification methods. The techniques are classified as techniques for the specification of external interaction and internal decomposition. The external interaction specification techniques are further subdivided into techniques for the specification of functions, behavior, and communication. After surveying the techniques, we summarize the way they are used in structured and object-oriented methods and indicate ways in which they can be combined. The article ends with a plea for simplicity in diagram techniques and for the use of formal semantics to define these techniques. The appendices show how the reviewed techniques are used in 6 structured and 19 object-oriented specification methods.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*tools*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*; *state diagrams*

General Terms: Design, Languages

1. INTRODUCTION

In the past 20 years, a large number of methods have been proposed for the specification of software systems. Until 1988, the proposed methods followed the structured approach and since that time, most proposals have followed the object-oriented approach. In the short time since 1988, at least 19 object-oriented methods have been proposed in book form and many more have been proposed in conference and journal papers. This survey puts these developments in perspective by reviewing the state of the art of structured as well as object-oriented methods. It identifies the underlying composition of structured and object-oriented software specifications, investigates in which respects object-oriented specifications differ essentially from structured ones, and points out opportunities for combining the two kinds of specifications. Most major structured and object-oriented specification methods are reviewed in the appendix, where "major" means that the method must have been described in a book rather than only in a conference or journal paper. The distinction between structured and object-oriented methods is artificial because some methods, such as JSD and SDL 92, occupy an intermediate position in this division and other methods, such as those of Shumate and Keller [1992] and Firesmith [1993], incorporate elements

Author's address: Department of Computer Science, University of Twente, Enschede, the Netherlands.

of both. Nevertheless, the division has a broad validity and it simplifies the discussion. Qualifications are made where necessary.

To keep the discussion manageable, we restrict ourselves to the specification of software at an external and conceptual level. These levels are defined in Section 2. With this restriction, what does a specification method include? We consider a software specification method to have the following components.

—*Techniques* for representing properties of software. Examples of such techniques are natural languages, formal languages, and diagram techniques.

—*Interpretation rules* for these techniques. For example, natural languages are interpreted as described by some dictionary and grammar, formal languages have a formal semantics, and diagram techniques have some intended meaning described in method handbooks.

—*Interconnection rules* for the techniques. These interconnection rules tell us how different techniques can be combined into a coherent specification of software and what this combined specification means. For example, in different methods, entity-relationship models are connected in various ways to dataflow models and to state transition models. Each of these ways is characterized by different interconnection rules that are, or should be, defined by the method handbook.

—*Heuristics* for the usage of the techniques. These heuristics should be consistent with the meaning of the techniques and the interconnection rules. Often they follow from these meanings. For example, if class specialization is interpreted as subsetting of class extensions, a simple heuristic to find specializations is to ask whether an instance of the subclass is always also an instance of the superclass. If specialization is interpreted in another way, for example, as reuse of class specifications, this heuristic may not apply but other heuristics will be valid.

We restrict our attention to semiformal techniques, by which we mean diagram techniques and techniques that use some form of structured natural language.

The structure of this article is as follows. In the rest of this introduction, we briefly discuss other surveys of structured and object-oriented specification methods. Section 2 introduces a framework in terms of which we present and analyze the methods. This framework is based upon the philosophy of systems engineering and is general enough to accommodate structured as well as object-oriented methods without distorting the characteristic philosophy of these methods. It allows us to present the techniques in Section 3 in a systematic manner and to disentangle in Section 4 the agreements and differences between structured and object-oriented methods. The actual survey of these methods is relegated to the appendices, so that readers can refer to those methods with which they are less familiar. Appendix A surveys 6 structured specification methods and Appendix B surveys 19 object-oriented methods in historical order of first publication in book form. The subsections of these appendices can be read in any order.

In Section 3 we review the techniques employed by the methods surveyed in the appendices. This is not a catalogue of all possible techniques that can be used in software specification. For example, Petri nets are not discussed and, as pointed out, formal specification techniques are left out of consideration. However, the coverage of the techniques used in the methods listed in the appendices is fairly complete. We do not discuss every variant of every technique but restrict ourselves to a representative of every technique "family." For example, we do not discuss all class modeling techniques but discuss the UML variant only. Other overviews of tech-

niques are given by Davis [1993] and Martin and McClure [1985].

Section 4 compares the way structured and object-oriented methods make use of these techniques, focusing on the interconnection rules and the characteristic heuristics used to find a specification. We identify some essential differences and agreements and point out unification opportunities. Section 5 winds up the survey with a summary of findings and directions for further research.

Throughout the article, we use a control system for a juice plant as a running example, adapted from an example given by Shlaer and Mellor [1992]. The control system controls the way juice is transported from storage tanks to cooking tanks, heated, and then transported to an operation where it is canned.

Since 1992, a number of surveys and comparisons of object-oriented methods have appeared. Fichman and Kemerer [1992] compare three structured with three object-oriented analysis methods along 11 modeling dimensions. These dimensions were identified by analyzing the six reviewed methods but they are not explained in detail. It is, for example, not clear why this would be a complete list of dimensions or whether all dimensions are at the same level of abstraction. Fichman and Kemerer do not present the techniques used in the methods in detail, as we do. They also discuss structured and object-oriented design methods, which we leave out of consideration. Sutcliffe [1991] lists four features of object-oriented models (abstraction, classification, inheritance, and encapsulation) and tabulates five object-oriented methods against these features. He also lists three features of structured models (functions, data, events) and tabulates eight structured methods against these features as well as against the object-oriented features. The discussion is very sketchy and there are no clear conclusions.

All other reviews ignore structured methods. Most of them give a list of features of object-oriented models and/or the object-oriented software development process, and tabulate some methods against these features. Taken separately, the lists look plausible enough, but different papers use different lists of features and none of them makes clear why the features they propose should be part of the list, or whether their list encompasses all essential features of object-oriented methods. Champeaux and Faure [1992] discuss a number of key object-oriented concepts such as inheritance, attributes, events, and the like, and then tabulate for each of 12 methods whether these concepts occur in the method. Monarchi and Puhr [1992] distinguish a number of features of the process and notation of object-oriented methods and then tabulate for each of 23 methods whether these features occur in the method. Most of the reviewed methods are described in papers; some are described in books. Monarchi and Puhr use a list of 14 notation features, including the ability to represent objects, relationships, attributes, behavior, communication, and so on. Fowler [1994] looks at seven object-oriented analysis and design methods and discusses a number of diagram techniques used in them. He then gives a number of guidelines for choosing an object-oriented method. Hodgson [1994] gives a brief history of structured and object-oriented methods without going into much detail. Hutt [1994b] and Hutt [1994a] contain a compilation of method descriptions, all according to the same format, submitted by those who invented the methods. Embley et al. [1995] draw up a list of 30 features of an object-oriented analysis notation and evaluate six methods on this list, again by tabulating features against methods. The features include the ability to represent objects with identity, state, and behavior, the ability to represent classes, relationships, aggregation, generalization, actions, triggers, and so on. Iivari [1995] compares six methods for object-oriented analysis on their ability to model the structure, function, and behavior of software systems. He con-

cludes that object-oriented methods are underdeveloped in the specification of external functions of the system as a whole (as opposed to the functions of individual objects). He also concludes that object-oriented methods are weak in guidelines to partition the system into subsystems. These conclusions agree with the conclusions we draw in this article.

This survey differs from all of these in that we give a detailed catalogue of techniques used in structured and object-oriented methods, and of the use of these techniques made by these methods. A second difference is that in our analysis (Section 4), we concentrate on the overall structure of software specifications and look for commonalities and differences in the reviewed methods with respect to this overall architecture. Our comparison framework is carefully argued from a small number of principles taken from systems engineering. We do not enter a discussion about which features should and should not be supported by a "true" object-oriented method. We take a nonpartisan approach by attempting to appreciate every method, structured or object-oriented, on its own terms and by trying to recognize opportunities for advancing the state of the art beyond the structured-object-oriented dichotomy by picking the best ideas from both approaches.

2. A FRAMEWORK FOR COMPARISON

The framework set out in this section is described and motivated in detail elsewhere [Wieringa 1996]. The basic concept in the framework of comparison is that of a system. A *system* is an assemblage of parts forming a complex or unitary whole that serves a useful purpose [Blanchard and Fabrycky 1990, pp. 1–2]. In this wide sense, organizations, pieces of furniture, thermostats, cars, and software products are examples of systems. It is crucial that the parts of a system interact in such a way that they cause the system as a whole to have a useful function for one or more entities in the environment of the system. An arbitrary collection of items with some interactions among the items is not necessarily a system. In order for a collection of interacting items to count as a system, the interactions among the items must produce a coherent and useful overall behavior.

In systems and software engineering one often reserves the term "system" for the entire constellation of equipment, software, and human procedures to be developed and refers to the software components of the system as "software products." Here we use the term in a very liberal way to indicate any collection of interacting items that serves a useful purpose, including hardware, software products, and software product components. However, we do restrict ourselves to systems that interact with their environment.

A system delivers a service to its environment by interacting with it. There are certain relevant ways in which we can describe these interactions. Interactions can always be partitioned into pieces that are useful for the environment, called *functions.* Examples from the juice plant controller are controlling the transport of juices from storage to cooking tanks, controlling the heating of the cooking tanks, and the like. Second, system interactions are by nature always *communications* with one or more entities in the environment of the system. For example, a juice plant controller communicates with storage tanks, cooking tanks, temperature sensors, and so on. Third, the way in which functions are ordered in time is called the *behavior* of the system. For example, a cooking tank must be heated only after it has been filled with juice, and heating involves switching the heater on and off in order to maintain the temperature for a period of time. Communication and behavior should be treated as orthogonal properties of functions. Communications are the way functions are ordered in space where behavior is the way in which functions are ordered in time. Taking the grand

view, the distinction between communication and behavior is similar to the distinction between space and time. Closer to home, but still taking a ball park view, the distinction is a basic property of algebraic process description languages, where operators to specify combinations of communicating processes are distinguished from operators to specify the dynamics of processes [Milner 1980, p. 61].

Functions, communications, and behavior are system properties. There are other kinds of system properties, such as security and user-friendliness, but these are not specified by the techniques reviewed in this survey and we leave these out of consideration.

We can describe properties at different levels of abstraction. For example, at the most abstract level, the interactions between a juice plant and its environment can be described as "provide canned juice to customers." This highest level description of the interactions is called the *mission* of the juice plant. At a more detailed level, we may describe functions such as "take order," "deliver juice," and "order ingredients," and at the most detailed level we may describe atomic transactions between the juice plant and its environment. This leads to the idea of a *refinement hierarchy* of behavior descriptions, in which the behavior of a system is described at increasing levels of detail. The lowest level of detail is that of the atomic pieces of behavior with which a system interacts with its environment. These are called atomic because intermediary states that occur during the behavior are not externally observable, or at least not considered to be externally observable. These atomic pieces of behavior are called *transactions.*

The parts of a system may themselves be systems that act together to produce the behavior of the system. This leads to the idea of an *aggregation hierarchy* of systems, in which systems at one level of aggregation have subsystems at the next lower level of aggregation and are themselves part of a compound system at the next higher aggregation level. There are two important ways to describe the decomposition of a system: we can specify its components and, for each component, we can specify the way it interacts with its environment to realize overall system behavior. The environment of a component consists of other components as well as the environment of the system. The specification of the interaction of the components can itself be divided into the specification of communication, functions, and behavior of each component. This pattern repeats itself at every level of aggregation. It follows that when we talk about properties, interactions, functions, and the like, we should always indicate the level in the aggregation hierarchy about which we are talking. In this survey, we assume that we have fixed a system of interest and talk about system functions, system communications, and the like, in contrast to component functions, component communications, and so on. To emphasize that system functions, and the like, are interactions with the environment, we also talk of external system functions, and the like.

Note that components can interact with the system environment. It is through these interactions that the system interactions are realized. Note also that the concept of a transaction, introduced previously, must be related to that of aggregation level. For each subsystem we can define transactions as subsystem functions whose internal states we consider unobservable. A system transaction is then realized as a process consisting of lower-level subsystem transactions.

The aggregation and refinement hierarchies are orthogonal. This can be represented visually by what Harel and Pnueli [1985] call the magic square of system development (Figure 1). Every point in this square represents a level of aggregation and a level of interaction refinement. The top-left point represents the highest aggregation level and the highest level of abstraction: the overall mission of the entire system.

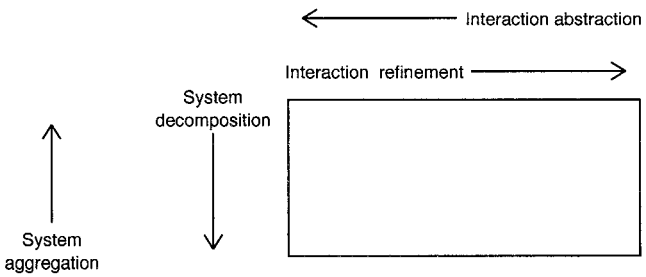

Figure 1. The magic square. The arrows point towards the direction of increase. To increase behavior refinement is to decrease behavior abstraction and to increase system decomposition is to decrease the aggregation level.

The bottom-right point represents the lowest aggregation level and the lowest level of abstraction: the atomic transactions of the lowest-level components of the system. The orthogonality of the two dimensions of the square means the following.

—At a given level of aggregation and abstraction, we can decrease the abstraction level at which we specify the interactions of a system at that level without decreasing the aggregation level. This is called *interaction refinement.* At the highest level of refinement (the lowest abstraction level), we specify the transactions of a system.

—Conversely, at a given level of aggregation and abstraction, we can decompose a system without decreasing the level of abstraction. This is called *interaction allocation* and *flowdown.* We allocate a system interaction to one or more components if we decide that these components will realize the system interaction. We flow down the system interaction if we specify what the component interactions are through which the system interaction is realized.

We can apply the idea of refinement to the three aspects of interaction (viz., functions, behavior, and communication). This gives us three species of refinement; function, behavior, and communication. Of these, function refinement is the most basic, because the system functions provide the *raison d'être* of the system: the system exists to provide functions to its environment. We therefore look at function allocation and flowdown in somewhat more detail.

Consider a system S at a certain aggregation level, whose functions are specified at a certain level of abstraction. This corresponds to a point in the magic square. Suppose that the functions of S at this level of refinement are function 1, . . . , function n and that we have identified components component 1, . . . , component m. We next decide for each component whether it plays a role in each function. This can be represented visually by means of a *traceability table* such as shown in Figure 2. A cross in an entry means that the component plays a role in the function, without yet indicating what this role is; absence of a cross indicates that this component plays no role in the realization of this function. A traceability table represents the allocation of functions to components. It is called a traceability table because it allows us to trace component functions to system functions and vice versa. If we add more information and indicate in each entry what the function is that the component should have in order to play this role in the system function, then we call the resulting table a *function decomposition table.* This represents the flowdown of functions to components. Allocation and flowdown are well-known techniques from systems engineering [Davis 1993,

	function 1	.....	.....	.....	function n
component 1	X				X
.....			X		
component m	X				

Figure 2. Function decomposition table.

p. 190]. The function decomposition table plays a central part in our analysis of methods.

Before we summarize our framework, we must make one more distinction. It is useful to distinguish a conceptual decomposition of a system from a physical decomposition. A *conceptual decomposition* is a decomposition whose partitioning criterion is defined in terms of the external environment of the system, which is the environment where the users of the system reside. (Users may be people or other systems.) The *physical decomposition* of a system is a decomposition defined in terms of the underlying physical components of the system, such as lower-level equipment or software components. The physical decomposition criterion does not refer to the external environment but to the underlying implementation environment. For example, an entity-relationship diagram represents a conceptual decomposition of a database. A relational database schema, in which each relation corresponds to an entity type or relationship in an entity-relationship model, is still a conceptual decomposition. A relational schema, in which the relations correspond to storage entities such as files, is a physical decomposition. In this schema, conceptual entities may have been split or merged in order to improve performance. Another example of a conceptual decomposition is a JSD model in which the components correspond to domain entities or to external system functions. In a physical decomposition, these are all mapped to the available physical resources and a scheduler may have been added to interleave processes that are conceptually concurrent.

The conceptual decomposition corresponds to the essential model of McMenamin and Palmer [1984] and the specification model of Cook and Daniels [1994]. It is a halfway station between the specification of external functionality and the internal physical decomposition. In the physical decomposition, issues are dealt with such as the distribution of resources, processing time, memory capacity, and error handling. The conceptual decomposition is a way to make the demands of external functionality explicit without yet worrying about these implementation decisions.

The relationship between the conceptual and physical decompositions is many–many. One conceptual component is implemented in one or more physical components, and one physical component implements one or more conceptual components. This can be represented by a second kind of traceability table, which sets conceptual against physical components. In this article, unless otherwise stated, the decompositions that we talk about are conceptual, not physical.

To summarize our framework, we have a number of kinds of system properties that we can specify. The following list relates these to the function decomposition table of the system.

—*System functions.* These are the useful pieces of behavior offered to the environment of the system, and corre-

spond to items in the top row of the table.

—*System behavior.* This is the behavior of the system over time. This concerns the way in which the functions in the top row are, can, or must be ordered in time. It thus concerns the entire top row.

—*System communication.* This concerns the communication of the system with external entities in its environment. It does not correspond to an aspect of the function decomposition table of the system, but it would correspond to a column of the function decomposition table of the environment, of which the system is one component.

—*Conceptual decomposition.* These are the conceptual components of the system, defined in terms of their meaning for the external environment, and correspond to the items in the leftmost column of the table.

—*Component functions.* These are the useful pieces of behavior offered by the components to their environment, and correspond to entries in the table.

—*Component behavior.* This is the way in which component functions are, can, or must be ordered in time, and corresponds with a row in the function decomposition table.

—*Component communication.* This is the way in which the components interact in order to realize the external functions, and corresponds to the columns of the function decomposition table. In each column, the component functions are listed that interact to realize the external function.

Due to the repetition in this pattern, there are only four different kinds of properties that we need specification techniques for:

—function specification techniques,

—behavior specification techniques,

—communication specification techniques, and

—decomposition specification techniques.

There are of course other properties, often called "nonfunctional," that can be specified, but the reviewed methods offer no techniques for these. In the next section we therefore use the preceding classification of techniques.

3. SPECIFICATION TECHNIQUES

3.1 Graphical Techniques

We start with a brief summary of the graphical structures encountered in the various techniques that follow. In general, there are only two kinds of graphical structures, graphs and Venn diagrams. A *graph* is a set of nodes connected by edges. (A node may be connected to itself.) Usually, each edge connects two nodes, but in a *hypergraph,* an edge may connect two or more nodes. In a *directed graph,* the edges have a direction. So in a directed hypergraph, each hyperedge has a collection of source nodes and a collection of destination nodes. In a *labeled graph,* edges and/or nodes may be labeled. A *tree* is a graph with one designated node called the *root* and in which from every node there is exactly one path to the root. The nodes with the longest paths to the root are called the *leaves* of the tree.

A graph can be drawn in many different ways, so that one must distinguish the graph from a presentation of the graph. Many methods annotate their graph presentations with additional comments and adornments that do not belong to the underlying mathematical formalism of a graph. If there are different kinds of nodes and edges, these are usually presented by different shapes. Many methods allow dangling edges that are connected to one node only and "dangle in the air" at the other side. This can be assimilated with the idea of a graph by allowing some nodes to be invisible in the presentation.

A *Venn diagram* is a collection of areas indicated by closed curves. We call these areas *blobs.* Hierarchy can be represented by a Venn diagram by enclosing one blob within another. Venn dia-

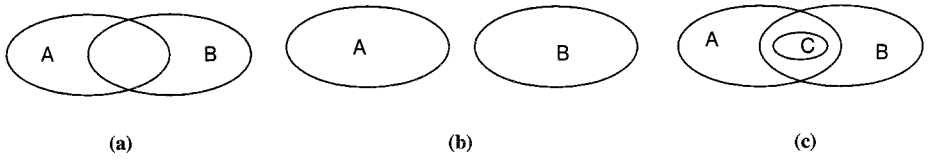

Figure 3. Higraphs versus Venn diagrams.

grams as such are not used in any of the reviewed techniques but the idea to represent hierarchy by enclosure is used in several techniques, such as SDL block diagrams and in the Statemate languages.

A *higraph* is a combination of a Venn diagram and a hypergraph, extended with the explicit labeling of areas and Cartesian products. The nodes of a higraph are blobs and these may be connected by hyperedges. Explicit labeling of areas means that in Figure 3, diagrams (a) and (b) are equivalent representations of the same higraph. In diagram (c), A and B have an intersection, called C. Since Venn diagrams do not require explicit labeling of areas, diagrams (a), (b), and (c) are different Venn diagrams. The Cartesian product A of blobs B and C is represented as shown in Figure 4. Higraphs were introduced by Harel [1988].

The following sections describe the use of these graphical structures in a number of specification techniques. In general, one cannot tell by looking at a diagram what it represents. The meaning of a diagram is given by a set of interpretation rules, which must be defined elsewhere. The interpretation rules are never specified together with the diagram but the diagram must be accompanied with an indication of which interpretation rules are intended to be applied. Without such an indication, the diagram is meaningless. In most cases, the interpretation rules for a particular kind of diagram are given in a book that describes the method. In some cases, such as the statechart language of Statemate, the interpretation rules are formal [i-Logix 1991a], and in some very rare cases, such as for SDL,

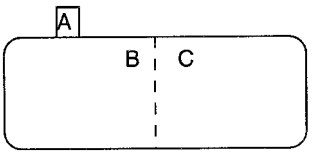

Figure 4. A higraph with a Cartesian product.

the interpretation rules are laid down in a standards document [ITU 1992].

3.2 Decomposition Specification Techniques

Decomposition specification techniques can be used to specify the items in the left-most column of the function decomposition table.

3.2.1 Entity-Relationship Diagrams. Entity-Relationship diagrams (ERDs) were introduced by Chen [1976] to represent the conceptual structure of data in a database system. An ERD is a labeled hypergraph in which the nodes represent types of entities and the hyperedges represent arbitrary relationships. The nodes of the hypergraph are presented by rectangles and the hyperedges by diamonds.

Figure 5 contains a fragment of ERD for a juice plant control system, derived from the specification of Shlaer and Mellor [1992]. The diagrams shows that there are two kinds of TANKs, STORAGE TANKs and COOKING TANKs. The hyperedge tank usage is called a specialization relationship and connects the node TANK with two other nodes, that are called its specializations. Each COOKING TANK is connected to one HEATER and each HEATER belongs to one COOKING TANK. A BATCH of juice is allocated to one COOKING TANK and

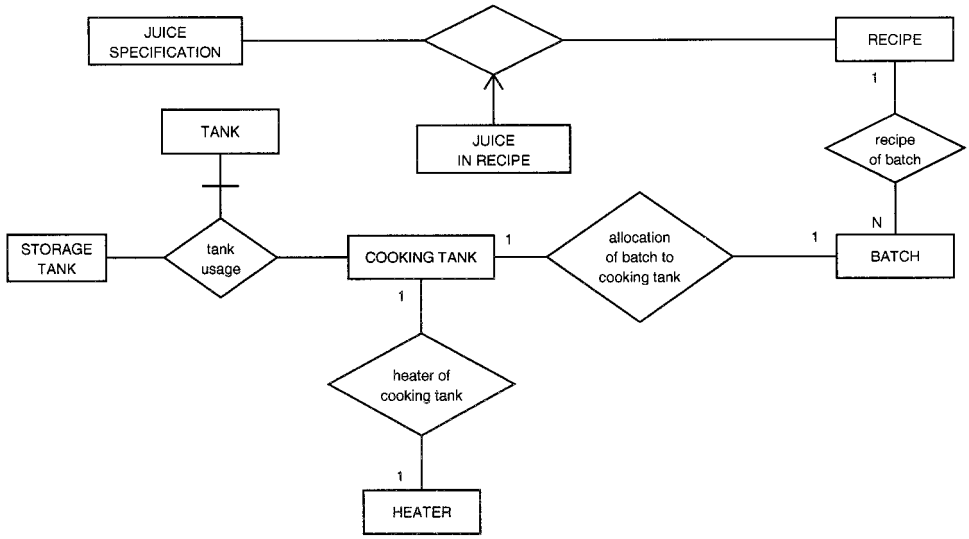

Figure 5. Fragment of an entity-relationship diagram of the data manipulated by a juice plant control system.

belongs to exactly one RECIPE. Each RECIPE is related to a JUICE SPECIFICATION. There is some special information connected to this relationship by means of the associative entity type JUICE IN RECIPE. The extra information is that there is a certain percentage of this juice in this recipe, but this is not detailed in the diagram.

A relationship node in an ERD can represent any kind of relationship. It may represent a communication link but it may also represent a visibility link, a permission, a part-of link, and so on.

From the start of their introduction by Chen, ERDs have been ambiguous, representing components in the environment of the software product or representing conceptual software components. This ambiguity arises from the fact that in database systems, where ERDs are mainly used, the conceptual structure of the data must correspond to the conceptual structure of the part of the world represented by the database. The same ambiguity can also exist in models of control-intensive systems: An ERD can represent the part of the world controlled by the system or it can represent conceptual objects inside the system. One cannot tell by looking at Figure 5 whether COOKING TANK represents real cooking tanks in the environment, or software objects in the control system. It follows that, in addition to the normal interpretation rules for a notation, an ERD must be accompanied by an annotation that tells the reader whether the ERD is to be interpreted in the real world or in the software.

3.2.2 Class Diagrams. There are many variations and extensions of the ERD technique. All object-oriented specification methods use some extension of the technique. We refer to this as a *class diagram,* but different OO methods tend to use different variations or extensions with different names, such as "information model," "object model," "object structure diagram," "static structure diagram," and so on. Figure 6 contains a class diagram following the conventions of the Unified Modeling Language (UML, Appendix B.19). The parallel with Figure 6 is clear. A noteworthy difference is that nodes now represent object classes, where objects have

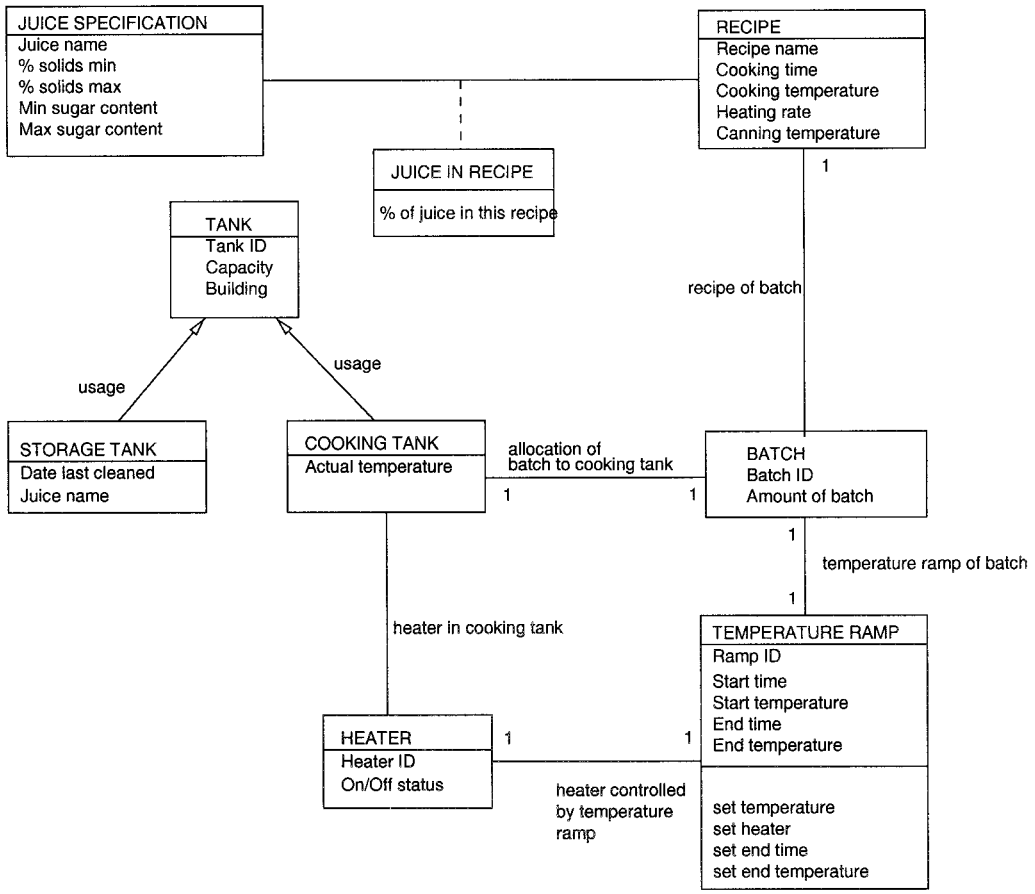

Figure 6. Fragment of a UML class diagram of the juice plant control system.

a local state represented by attributes, and they have local operations that may change those attributes. Entities specified by ERDs also have attributes, but in structured analysis they do not have local operations. A node in a class diagram may be labeled with some or all attributes and operations of the objects of this class.

A second important difference is that the class diagram contains an additional object class, TEMPERATURE RAMP, which has no counterpart in the ERD. This is not a difference in diagram techniques but in methodology. The task of instances of this class is to control the heating process of a cooking tank, something that would be modeled by a control process in dataflow models.

3.3 Communication Specification Techniques

Communication specification techniques show how the conceptual components interact to realize external system interactions. This corresponds to the columns in the function decomposition table.

3.3.1 Dataflow Diagrams. A dataflow diagram (DFD) is a labeled directed hypergraph in which the nodes represent functions and the edges dataflows between functions. It is used in structured analysis to specify the functions of a system. There are three kinds of nodes in a DFD that represent functions, data stores, and external entities. The functions and data stores are defined in terms of their meaning for the

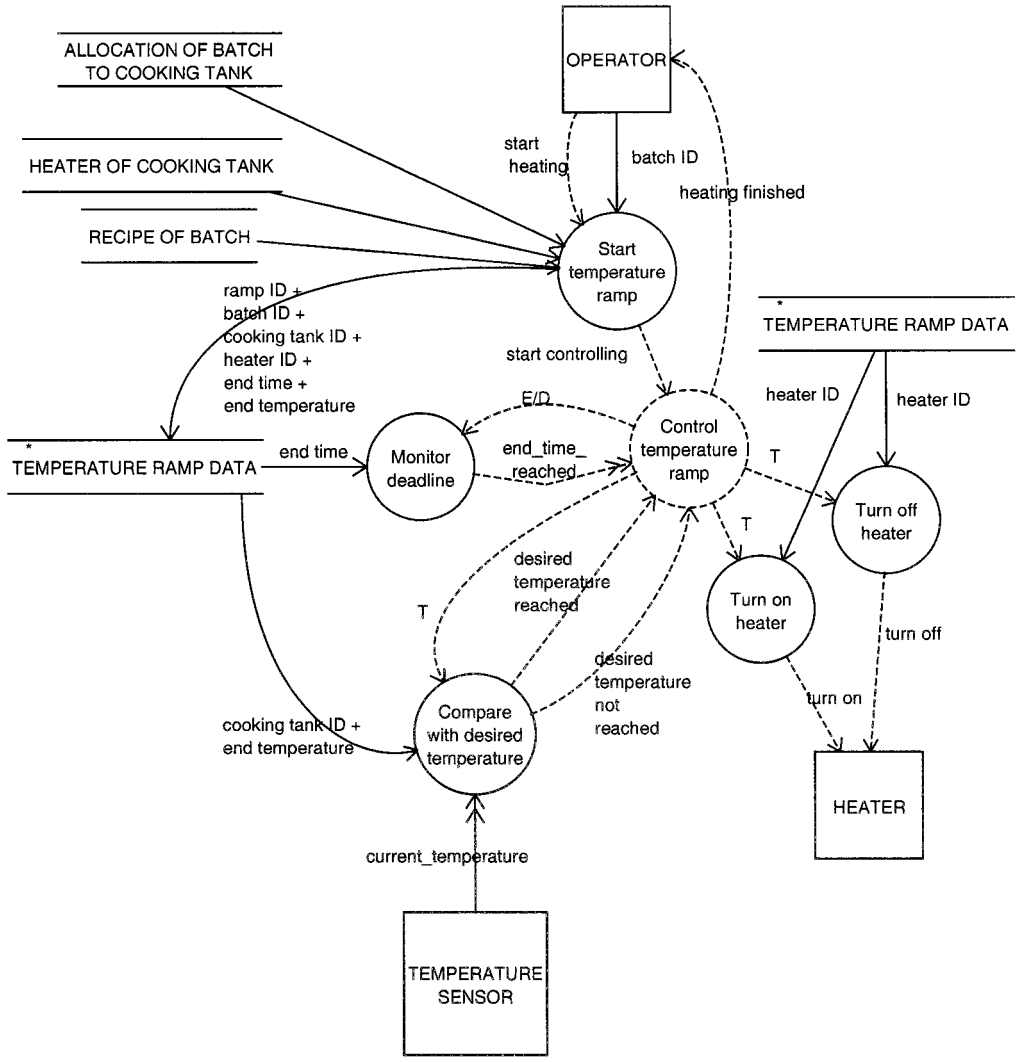

Figure 7. Fragment of a dataflow diagram of the functions of the juice plant control system.

external interactions of the system, so they are conceptual (not physical) system components.

Figure 7 shows a part of a DFD of the juice plant control system, following the YSM convention [Yourdon Inc. 1993]. External entities in the environment of the control system are represented by squares, functions are represented by circles, memory by two parallel lines, and dataflows by arrows. The asterisk in the label of the data store TEMPERATURE RAMP DATA means that this data store has multiple representations in this diagram.

Most versions of structured analysis, including the Yourdon Systems Method (YSM, Appendix A.2), distinguishes two kinds of functions, called *data processes,* represented by unbroken circles, and *control processes,* represented by dashed circles. Data processes transform data; control processes transform the mode of behavior of the system. For example, Figure 7 shows that there is a control process Control temperature ramp and a

number of data processes to turn a heater on and off, and so on. This control process corresponds to the TEMPERATURE RAMP object class of Figure 6. We discuss the methodological difference between these two modeling approaches when we look at decomposition criteria in Section 4.2.3.

A second distinction made by YSM is that between *instantaneous processes,* whose activity lasts only one instant of time, and *continuing processes,* whose activity lasts a period of time. There is no visual convention to distinguish the two kinds of processes. In Figure 7, Monitor deadline is a continuing data process and Control temperature ramp is a continuing control process. All other processes in the figure are instantaneous. Control processes are usually continuing processes. An instantaneous control process has no memory and can always be specified by a decision table that transforms a number of discrete binary input flows into a number of discrete binary output flows.

The diagram shows the data interfaces between the functions (unbroken arrows) as well as the event interfaces (dashed arrows). The double-headed arrow represents a time-continuous flow. A continuous dataflow contains data values for periods of time. A continuous event flow contains Boolean values that may be present over periods of time. There are some special symbols that can appear as labels of event flows. For example, T, E, and D stand for triggering, enabling, and disabling a data or control process, respectively.

The data processes in Figure 7 must be specified in structured analysis by textual specifications that define the relation between their inputs and outputs. This is not treated in this article. The control process behaves as a finite state machine and must be specified by means of a state transition diagram or table (Section 3.5.5).

All versions of dataflow modeling since DeMarco [1978] use hierarchical DFDs to reduce the complexity of a large dataflow diagram. Any part of a large diagram can be abstracted away by replacing it by a process group, which has the same visual presentation as a data process (an unbroken circle). The process group is specified separately by showing the DFD portion for which it stands. The intention is that a process group represent a coherent group of software functions. This convention is well suited to the representation of a functional decomposition of the system, discussed in Section 4.2.3.

3.3.2 Context Diagrams. A context diagram is a graph in which one node represents the system and the other nodes represent important systems in its environment. The edges represent communications among the systems. A context diagram will at least show the systems with which the system under development communicates, but it may show other systems, if these are necessary to understand the behavior required of the system under development [Jackson 1995].

In structured analysis, a context diagram only shows the systems that the system communicates with, called external entities, and the edges represent dataflows between the system and its external entities (Appendix A.2). In other words, in structured analysis, a context diagram is a dataflow diagram. Large context diagrams may be split into a number of partial context diagrams, one for each major system function.

3.3.3 SADT Activity Diagrams. An activity diagram is a directed hypergraph in which the nodes represent activities and the edges represent flows of data, matter, or energy between activities. Activity diagrams are used in SADT to specify a conceptual decomposition of a system into a number of cooperating activities (Appendix A.1). The technique was introduced by Ross [1977]. Figure 8 shows an activity diagram with three activities, represented by boxes with rounded corners. Arrows entering a box from the left represent

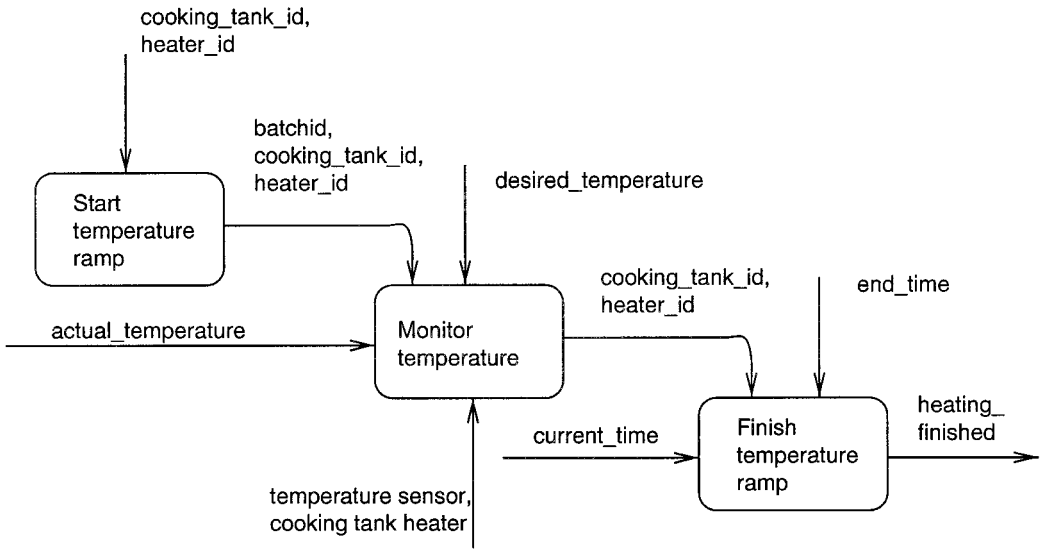

Figure 8. Activity diagram.

input data, arrows leaving to the right represent output data, arrows entering from above represent control information, and arrows entering a box from below represent mechanisms by which the activity is to be performed. Activity diagrams can be structured hierarchically, so that one activity box can be decomposed into a lower-level activity diagram. The decomposition hierarchy does not represent a physical decomposition of the system under description, but is a conceptual decomposition of the tasks to be performed by the system. This convention is well suited to the representation of functional decomposition. In contrast to DFDs, activity diagrams have only one kind of component and the interfaces between the activities distinguish input, output, control, and mechanism.

3.3.4 Statemate Activity Charts. Activity charts are higraphs without intersections or Cartesian products that are used in Statemate to represent external system behavior (Appendix A.3). They are variants of DFDs with a different syntax but with a similar meaning. Statemate distinguishes external activities, regular activities, control activities, and data stores, which correspond to external entities, data processes, control processes, and data stores in DFDs, respectively. Regular activities are represented by rectangles with sharp corners, control activities by rectangles with rounded corners, and data stores by rectangles in which the vertical edges are dashed. Data and event flows between activities are represented by directed hyperedges. In Figure 9, D is a control activity, B a data store, and X and Y are external activities. The other nodes in the diagram are regular activities.

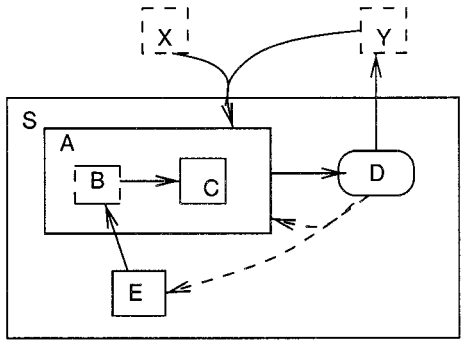

Figure 9. An activity chart.

The hierarchy of activities is repre-

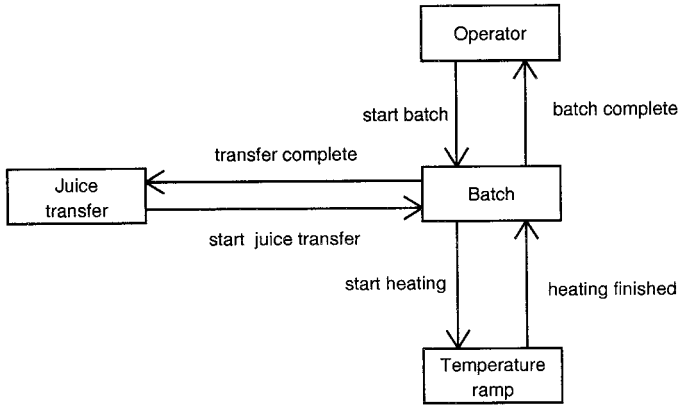

Figure 10. Fragment of an object communication diagram.

sented by means of inclusion of rectangles. Control activities are always subactivities of regular activities, and a regular activity can have at most one control activity as an immediate component. Control activities are specified by extended finite state machines, specified in Statemate by means of statecharts (Section 3.5.7). Extended finite state machines are finite state machines extended with variables that can be updated. Control activities can thus maintain a state in their variables, can perform data processing (testing and updating the variables), and can contain control.

3.3.5 Object Communication Diagrams. An object communication diagram (OCD) is a directed graph in which the nodes represent object classes and the edges represent possible object communications. The technique is used in the Shlaer–Mellor method to represent object communications. Figure 10 shows a fragment of an OCD based upon an example given by Shlaer and Mellor [1992]. For the sake of the example, the diagram represents a slightly different situation than that shown by the DFD in Figure 7; the start heating message is sent to the temperature ramp object not by an operator but by a batch object.

The diagram shows external entities (operator) as well as object classes. Edges represent possible communications. For example, the edge start batch shows a possible communication between an instance of Operator and an instance of Batch. Temporal ordering of communications is not represented.

3.3.6 JSD System Network Diagrams. A system network diagram (SND) is a directed graph in which the nodes represent processes and the directed edges represent communications. SNDs are used in JSD to specify system functions (Appendix A.6). Figure 11 shows an SND with three processes, represented by rectangles, and two communication connections. SV represents a state vector connection, which is a connection through which a reader process (TEMPERATURE RAMP CONTROL) reads the state of another process. D represents a data stream connection, which is a queue through which a sender sends messages to a receiver. The other circles represent queues through which the system receives messages from its environment and sends messages to its environment. The external entities that send and receive these messages are not represented in an SND. In JSD, each of the three processes in the SND would be an extended finite state machine, specified by means of a process structure diagram (Section 3.5.2). JSD allows these processes to have local state variables (called attributes) and to test and update these variables.

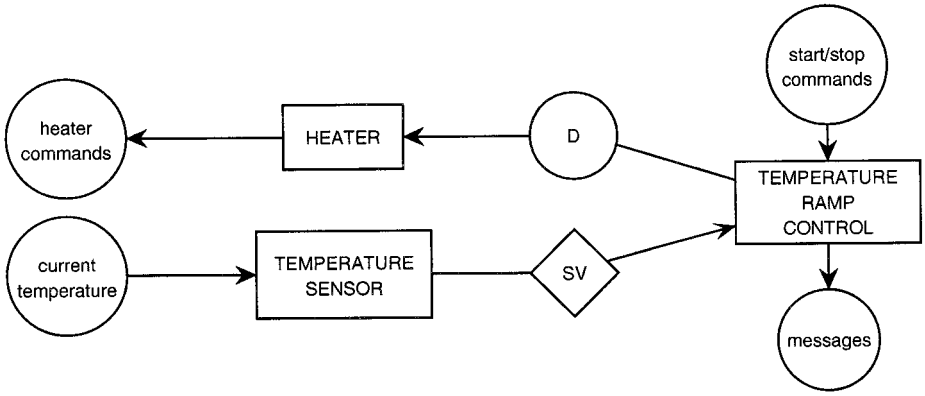

Figure 11. A system network diagram.

SNDs are similar to DFDs in that both represent the system as a network of communicating components. However, SNDs recognize only one kind of component, a process that maintains a state and has behavior over time. DFDs contain three kinds of components, that maintain a state (data stores), update it (data processes), and have a behavior over time (control processes). This is an important technical difference between the diagram techniques that is related to the concept of a state machine. In DFDs, control processes can do no data processing or data storage, whereas in SNDs, the processes can store, test, and update data. Activity charts occupy an intermediate position in this spectrum because they contain regular activities and data stores just as DFDs do, but their control activities can store, test, and update data as well. We return to this in the discussion of extended finite state machines in Section 3.5.4.

Another difference between the diagram techniques is that SNDs are flat whereas DFDs are hierarchical. Typically, one SND is specified for each external system function. From a methodological point of view this corresponds to the heuristic of event partitioning in structured analysis in which one DFD is drawn for every event-response pair of the system [McMenamin and Palmer 1984]. This means that for every column in the function decomposition table, a diagram is drawn. We elaborate on this in our discussion of decomposition criteria in Section 4.2.3.

The nodes in Figure 11 represent software components. TEMPERATURE SENSOR and HEATER represent entities in the part of the world controlled by the software, which is why we call them surrogates. They are represented as entity types in the ERD of Figure 5 and as object classes in the class diagram of Figure 6. TEMPERATURE RAMP CONTROL is a function process whose task it is to enforce certain behavior on the external entities. This corresponds to a control process in the DFD of Figure 7 and to an object class in Figure 6. The distinction between surrogates and function processes is an important methodological difference between functional decomposition and object-oriented decomposition. This too is discussed in more detail in Section 4.2.3.

3.3.7 SDL Block Diagrams. A block diagram is a graph in which the nodes represent *blocks* and the edges represent *channels* through which the blocks can communicate. They are used in SDL to partition a complex system into simpler parts (Appendix A.4).

Blocks communicate by sending signals through channels. Signals traveling along a channel may experience delay. A channel may carry signals in both ways. Figure 12 shows a simple block

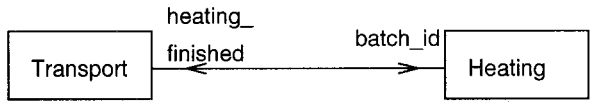

Figure 12. A simple block diagram.

diagram. The Transport block can send a batch_id to the Heating block, that can send a heating_finished signal back to the Transport block. When used to represent a system decomposition, these blocks represent system components. They are conceptual components, because they are defined in terms of their meaning for external system behavior.

A block can be decomposed into a lower-level block diagram, provided that the interface of a block is equal to that of its decomposition. Hierarchical diagrams can be used to specify different levels in an aggregation hierarchy, called service layers in SDL [Belina et al. 1991; ITU 1993]. Each leaf block in a block hierarchy contains one or more processes. Processes cannot be part of nonleaf blocks. This means that a block either has subblocks or it contains processes, but not both.[1] Each process is an extended finite state machine, specified by an SDL state diagram (Section 3.5.8). The processes within one leaf block communicate with each other by means of signal routes. The difference with channels is that signals experience no delay when traveling through a signal route, whereas they may experience delay when traveling through a channel. The behavior of a leaf block is the result of the behavior of its component processes. The behavior of a higher-level block is the result of the behavior of its component blocks.

Block diagrams are similar to DFDs and activity charts in that they all represent a hierarchy of system functions and their communications. An important difference is that in a block diagram, control is only specified at the leaves of the hierarchy, whereas in DFDs and activity charts, control may be specified at any level in the hierarchy.

A second important difference appears if we flatten both hierarchies. A flattened DFD or activity chart consists of a number of communicating control processes, data processes, and data stores. A flattened block diagram consists of SDL processes, which are extended finite state machines, just as control activities in activity charts and processes in SNDs are. Extended finite state machines contain control (a finite state machine), data processing (tests and updates), and data storage (variables).

A third difference between block diagrams and DFDs is that a channel or signal route combines the features of data and event flows: it may carry a single signal, like an event flow, or a signal together with data values, like a data flow.

Block diagrams are similar to SADT activity diagrams in that they both represent a collection of communicating activities that can be decomposed hierarchically. However, activity diagrams do not allow the inclusion of control and have no precise semantics for communication, as block diagrams do.

The idea of block diagrams is a natural one and it occurs in several object-oriented methods with a telecommunication background. For example, in OOSE, design components are called blocks [Jacobson et al. 1992, pp. 224–227]. In the commercialized variant of OOSE called Objectory, the term "block" has been replaced by the term "design object" [Objectory AB 1995a, p. 88]. See

[1] This simple picture is complicated by the possibility of combined block specifications, which from one point of view are leaf blocks used to specify the behavior that must be implemented, and from another point of view are nonleaf blocks, decomposed into lower-level blocks that implement this behavior.

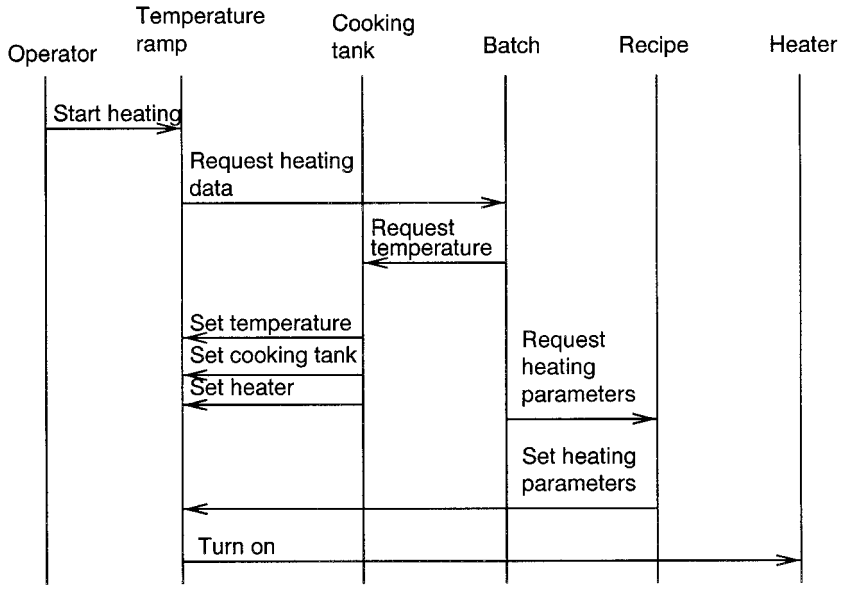

Figure 13. A sequence diagram.

also Appendix B.5. Block diagrams are also closely related to actor class diagrams in ROOM (Appendix B.11). An actor class diagram is a block diagram in which processes can be defined for blocks at each level of the hierarchy. The blocks in these diagrams are called actors.

3.3.8 Sequence Diagrams. A sequence diagram represents a particular sequence of messages exchanged between a number of entities. In the telecommunications area, they have been standardized as message sequence charts [ITU 1994]. They are used in combination with SDL specifications to illustrate sequences of messages between a system and its environment. They were introduced into object-oriented modeling by Objectory [Jacobson et al. 1992], and since then have become popular in object-oriented methods to represent communications between objects.

Figure 13 shows a sequence diagram that represents the communication among a number of objects in response to the operator message to start heating. The communicating entities are represented by vertical lines. The downward direction represents the advance of time. Horizontal arrows represent messages. Sequence diagrams can be extended with conventions to represent timeouts, global conditions across different entities, delayed message reception, and the like.

A sequence diagram shows one particular communication sequence in one run of the system. This stands in contrast to the communication specification techniques reviewed so far, which specify properties of all possible runs of the system. A second property of sequence diagrams is that they show communication as well as behavior. They could therefore also have been listed as behavior specification techniques.

3.3.9 Collaboration Diagrams. A collaboration diagram is a directed graph in which the nodes represent communicating entities and the edges represent communications. The edges are numbered to represent the ordering of communications in time. Figure 14 contains a collaboration diagram that represents the same message sequence as Figure 13. Collaboration diagrams have been

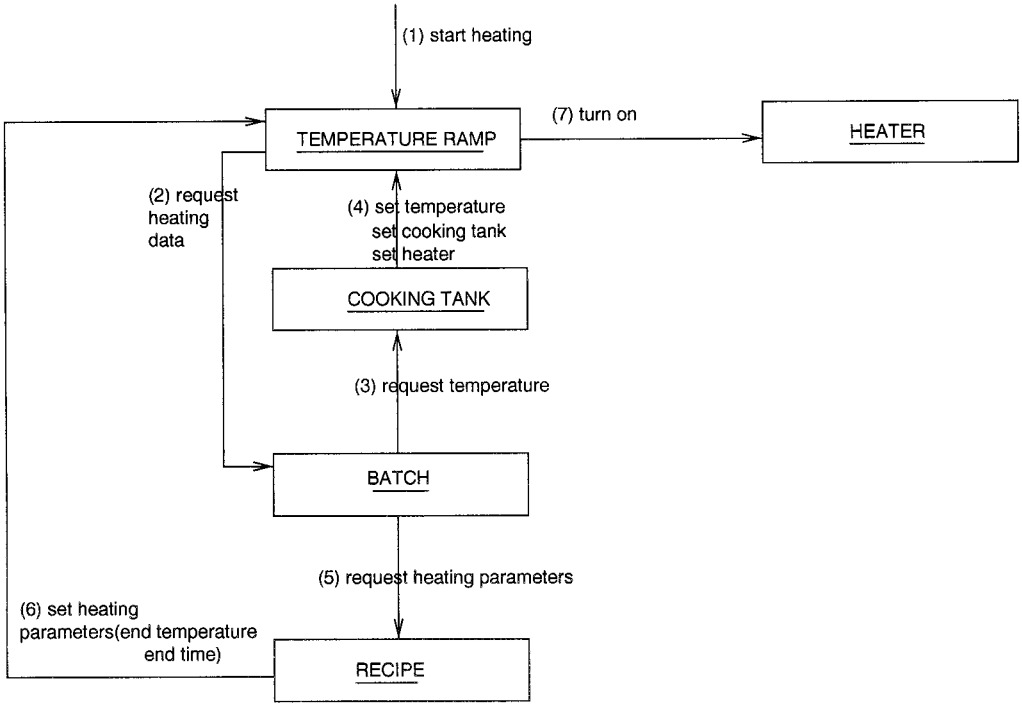

Figure 14. A collaboration diagram.

introduced by Wirfs-Brock et al. [1990] and have, in various guises, become popular in object-oriented methods to represent message exchanges between objects. In the UML (Appendix B.19), they can be extended with annotations that represent dataflows between the communicating objects and various adornments that represent the way in which the communications are implemented.

Collaboration diagrams differ from DFDs, activity diagrams, activity charts, and block diagrams because their nodes represent objects, not activities. This also sets them apart from object communication diagrams, because those diagrams represent object classes, not individual objects. In addition, just as do sequence diagrams, collaboration diagrams represent the sequence of messages in one particular scenario whereas all other communication diagrams represent possible communications in all possible scenarios.

3.3.10 Summary. We can distinguish two kinds of diagrams to represent communication; diagrams that show communication sequence (sequence diagrams and collaboration diagrams) and diagrams that show a set of possible communications without indicating any sequence (all other diagrams). Diagrams that show communication sequence illustrate communications as well as behavior in one particular run of the system. The other diagrams show communication only and do not refer to a particular run of the system.

A second distinction to be made concerns the kinds of things that can communicate. We have seen the following kinds of conceptual components.

—Finite state machines. Control processes in DFDs are specified by finite state machines.
—Extended finite state machines. These contain control, local variables, and tests and updates of these variables.

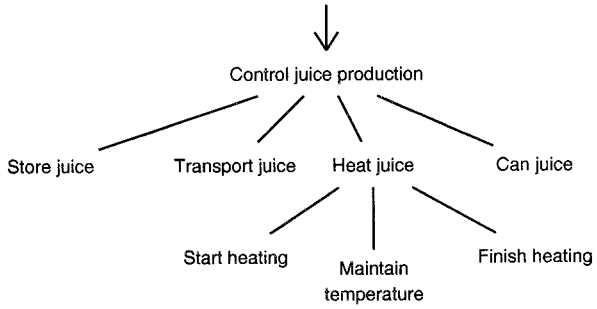

Figure 15. Part of a function refinement tree of a juice plant control system.

Control activities (activity charts), JSD processes, and SDL processes are examples. The objects in sequence and collaboration diagrams can also be viewed as extended finite state machines.

—Data processing activities. Examples are data processes (DFDs), activities represented by SADT activity diagrams, and regular activities (activity charts). All data processing performed by SDL blocks is ultimately part of SDL processes, which are extended finite state machines.

—Data stores. These are present in DFDs and activity charts.

In addition to these conceptual components, DFDs and activity charts also represent the external entities with which the system communicates.

Complexity reduction by means of hierarchy is possible in DFDs, activity charts, and block diagrams. Hierarchy may be represented in a Venn diagram-like way by enclosure of areas or implicitly, by a naming convention.

There is a considerable range in the possible properties of communication: synchronous or not, delay or not, reliable or not, unidirectional or bidirectional, and so on. The reviewed techniques should define a semantics for the notations so that it is clear what kind of communication is represented. Without such clarity, the notation may be used to give a rough sketch of vague ideas but not to communicate a meaning unambiguously to others. There is considerable difference among the techniques on the matter of semantics. SADT activity diagrams hardly have any communication semantics; DFDs and object communication diagrams are slightly less ambiguous. Petersohn et al. [1994] define different possible semantics for DFDs. SNDs have a better defined semantics, but there is nothing near a formal specification of this. SDL block diagrams and activity charts have precisely defined communication properties. The semantics of communication in block diagrams is formalized in an ITU standard.

3.4 Function Specification Techniques

Function specification techniques are used to specify the external functions of a system, which are the items in the top row of the function decomposition table, or the functions of system components, which are the items in each row of the table.

3.4.1 Function Refinement Trees. A function refinement tree represents system behavior at an increasing level of detail at the same level of aggregation. Figure 15 shows part of a function refinement tree of a juice plant control system. A function refinement tree increases the level of detail along the horizontal dimension of the magic square. The highest level of detail at a given level of aggregation is that of atomic functions, that is, transactions. The leaves of a fully grown function refinement tree thus represent transactions.

Function: Finish heating.

- **Event:** Temperature ramp completed.
- **Meaning**: Required heating time of cooking tank is over.
- **Desired response**: Switch off heater, start canning.
- **Detection mechanism**: Time
- **Response time**:
- **Data flows**: Tank identifier.
- **Event flows**: —
- **Stored data used**: Tank identifier and recipe.
- **Event detection process**:
- **Event response process**:
- **Used in control process**:

Figure 16. Event-response specification for a function of the juice plant control system.

We should stress that function refinement and system decomposition are orthogonal. Figure 15 is compatible with many different system decompositions, only one of which is a decomposition into modules that correspond to system functions. Other possible decompositions include subject domain partitioning, in which the conceptual components correspond to entities in the represented or controlled part of the world.

We cannot tell by looking at a tree diagram whether it is a function refinement tree or a system decomposition tree. Thus, interpreted as a system decomposition tree, Figure 15 would represent a functional system decomposition of the juice plant control system into functional modules. This is another example of the importance of annotating the diagram not only with the interpretation rules to be used, but also with an indication of the domain in which the diagram is to be interpreted.

In information engineering, a function refinement tree is used to represent a hierarchical decomposition of business functions of an organization independently from the actual organization structure (Appendix A.5). In product development, function refinement trees are used to represent the functions of a product in a hierarchical manner. If a function refinement tree is used together with a hierarchical diagram technique such as DFDs, SADT activity diagrams, Statemate activity charts, or SDL block diagrams, then the tree can be made to correspond to the hierarchical decomposition of the diagram.

3.4.2 Event-Response Specification. A simple way of specifying the functions of a system is to make a list of events to which the system must respond and write down the desired response for each event. Other properties can also be noted, such as where the event comes from, where the response goes to, what the desired response time is, and so on. It may also be useful to add information that relates the event-response pair to other parts of the specification, such as an indication of the conceptual components that handle the event.

Figure 16 shows an example event-response specification of a function ac-

cording to YSM (Appendix A.2). The function finish heating is triggered by the temporal event temperature ramp completed. This is called a *temporal event* because it consists of a significant moment in time, namely, the occurrence of the desired endtime for heating a cooking tank. This is noted in the specification by saying that the detection mechanism is time. The other kind of event that can trigger a system function is an *external event,* which is generated by an external entity. For external events, the external entity that generates it must be noted. The other properties of the function listed in Figure 16 only make sense when the specification contains a dataflow model. This is always the case in YSM but in other methods, one would want to specify other kinds of properties instead. For example, in object-oriented methods, one could specify which objects handle the event and which objects produce the response.

Event-response pairs are functions of the system. If event-response specifications are combined with a function refinement tree, they should thus correspond to nodes in a function refinement tree. Event-response pairs may or may not be transactions. For example, the response of a controller to the request to heat a cooking tank is to heat the cooking tank and this takes a significant amount of time, with intermediary states that are externally observable. However, the response of the controller to the event that the desired temperature is reached is to switch off the heater, and this conceptually takes place at the same instant as the event occurrence. Event-response pairs thus may or may not correspond to the leaves of the function refinement tree, and there may be refinement relations between event-response pairs.

3.4.3 Declarative and Imperative Specifications

Each function is executed when the system is in some state and leaves the system in a possibly different state. It may accept input from the environment and may produce output sent to the environment. The effect of the function can be specified textually in two ways, declaratively or imperatively. In a *declarative specification,* we describe pre- and postcondition pairs, where a precondition is a condition on the input and system state at the start of executing the function and the postcondition is a condition on the output and the system state after the execution of the function. In an *imperative specification,* we describe the activities to be performed to get from the input and initial system state to the output and resulting system state. There are a large number of semiformal and formal languages to write declarative or imperative specifications. We do not review these here but illustrate the idea by using natural language.

Figure 17 shows a declarative specification of the data process Start temperature ramp of the DFD in Figure 7. The outputs of this data process are Booleans and we use their names to indicate postconditions. The idea is that if the precondition is true for the input values, then the data process guarantees that the postcondition is true. If the precondition is false, then nothing is guaranteed. Preconditions and postconditions come in pairs, of which the preconditions are preferably mutually exclusive and jointly exhaustive of all possible cases. If they are not mutually exclusive, then the postconditions of overlapping preconditions must be mutually consistent. If they are not jointly exhaustive, then the effect of the function in some states is not specified.

Figure 18 shows an imperative specification of the same process. It reads the dataflow input, collects the necessary data from three data stores, and writes a record to another data store. The advantage of an imperative specification is that, written down in a suitable language, it leads to an executable specification that can be validated at an early stage of development. Its disadvantage

```
Event flow input: start heating
Data flow input:batch ID
Data store input:ALLOCATION OF BATCH TO COOKING TANK
HEATER OF COOKING TANK
RECIPE OF BATCH
Event flow output: start controlling
Data store output:TEMPERATURE RAMP DATA

Precondition 1:
batch ID occurs exactly once in ALLOCATION OF BATCH TO COOKING TANK
and allocation of batch is cooking tank ID
and recipe for batch ID occurs in RECIPE OF BATCH with end time and end temperature
Postcondition 1:
new(ramp ID) + batch ID + cooking tank ID + heater ID + end time + end temperature
exists in TEMPERATURE RAMP DATA

Precondition 2:
batch ID does not occur exactly once in ALLOCATION OF BATCH TO COOKING TANK
Postcondition 2: error

Precondition 3:
recipe for batch ID does not occur in RECIPE OF BATCH temperature
Postcondition 3: error
```

Figure 17. A declarative specification.

```
Event flow input: start heating
Data flow input:batch ID
Data store input:ALLOCATION OF BATCH TO COOKING TANK
HEATER OF COOKING TANK
RECIPE OF BATCH
Event flow output: start controlling
Data store output:TEMPERATURE RAMP DATA

If batch ID occurs exactly once in ALLOCATION OF BATCH TO COOKING TANK
then get cooking tank id from ALLOCATION OF BATCH TO COOKING TANK;
     get heater ID from HEATER OF COOKING TANK;
     if recipe for batch ID occurs in RECIPE OF BATCH
     then get end time and end temperature from RECIPE OF BATCH;
          create ramp ID;
          update TEMPERATURE RAMP DATA
          with ramp ID + batch ID + cooking tank ID + heater ID + end time + end temperature
     else error;
else error
```

Figure 18. An imperative specification.

is that it specifies a process by giving an implementation for it. For example, the implementation in Figure 18 gives a particular order of reading the data stores. This does not decrease the designer's freedom to choose another implementation of the same process but in general what exactly counts as the "same" process is a moot point. To find an equivalent implementation of the same process, one needs a declarative specification that says what the input-

output relation is that must be realized by the different implementations.

The advantage of a declarative specification is that it is completely implementation-independent. However, its disadvantage is that it is an underspecification. The implementor has to figure out what is changed as a result of the function and what remains the same. For example, the declarative specification of Figure 17 says that if precondition 1 is satisfied, then after an execution of the function a new record exists in the store TEMPERATURE RAMP DATA. This does not imply that the rest of the data store remains unchanged. It is a constraint on the next state of the store that can be satisfied in many different ways. The new record may be added to the existing records, or it may replace any or all of them. The declarative semantics of new will not tell this unless we add the frame assumption that whatever cannot be inferred from the postcondition to change remains the same. It is not a trivial matter to determine this, especially if the postcondition contains disjunctions or negations and implies derived updates. Borgida et al. [1995] summarize the frame problem for declarative software specifications. Imperative specifications do not have the frame problem, because they specify the change in terms of a sequence of elementary changes whose effect on the variables is completely specified. For example, in any specification language, the semantics of the create statement would be that a record is added to TEMPERATURE RAMP DATA. This is not a constraint that can be satisfied in various ways; it is an instruction that yields a definite result.

Whether one needs declarative or imperative specifications, or both depends upon what one wants to do with a specification. Declarative and imperative function specifications may be used to specify external system functions, for example, as part of event-response pairs, or component functions, such as object operations. They may also be used to specify state transitions in extended state machines, as discussed in Section 3.5.4.

3.4.4 Use Case Diagrams. Use cases have been introduced by Jacobson et al. [1992] in OOSE to represent external system functionality (Appendix B.5) and have been adopted since then by several other methods and notations. A *use case* is an interaction between the system and an external entity that has a use for that external entity. The interaction need not be atomic. In the terminology of this article, a use case is the same as an external system function. Use cases can be described by a narrative text, or more formally by a specification of pre- and postconditions of the use case. For complex use cases, Jacobson et al. [1992] recommend the use of a state transition diagram. An overview of the use cases can be given by a *use case diagram,* that shows for each use case which external entities are involved in it. Figure 19 shows a fragment of a use case diagram of the juice plant control system. Figure 20 gives an example of a specification of the heat the cooking tank use case, following the Fusion convention [Coleman 1996]. Use case specifications may be supplemented with sequence diagrams that illustrate the sequence of message exchanges.

A use case diagram actually shows system functions as well as external communications, and could also have been listed in the section of communication specification techniques. Like context diagrams, they show communications between the system and its external entities. The difference with context diagrams is that a use case diagram shows system functions and their communications, whereas a context diagram shows dataflows between the system and external entities. One can, however, draw partial context diagrams, one for each major system function [Yourdon Inc. 1993, p. 170]. Use case diagrams differ only marginally from these. If a use case diagram is combined with a function refinement tree, then the use cases should corre-

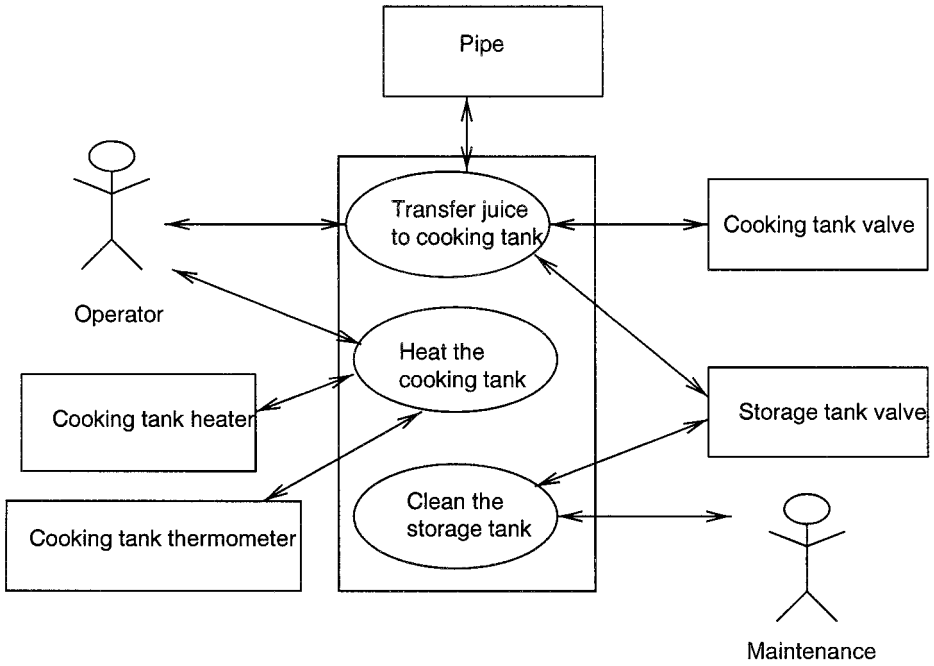

Figure 19. Fragment of a use case diagram of the juice plant control system.

spond to nodes in the diagram. Often, these nodes will not be leaves of the tree, because use cases generally are not atomic functions.

3.4.5 Summary. Each of the reviewed techniques draws attention to a different aspect of external system functionality. A function refinement tree relates the overall mission of a system to its functions down to its atomic transactions. For each function, it shows why the system must have that function—to realize its mission. Use case diagrams summarize the communications that take place during particular functions. All functions can be specified as event-response pairs but it is most efficient to do this only for atomic functions, that is, transactions. A context diagram can be used to specify the data exchanges between the system and its environment. One may choose to draw one partial context diagram for each use case, one for each major system function.

There is no need for formality in function refinement trees, context diagrams, or use case diagrams. The important point in using these is to follow good heuristics and not to put more meaning in these diagrams than is intended. The specification of event-response pairs is in need of a bit more formality. In particular, the specification of the effects of events must be carefully specified, declaratively or imperatively. When investigating a method, one should look carefully at the possibility of ambiguity in the specification technique used for this. This is important, because customers should get the system functions for which they thought they asked.

3.5 Behavior Specification Techniques

Behavior specification techniques can be used to show how functions of a system or of its components are ordered in time. This corresponds to the rows of the function decomposition table.

3.5.1 Process Graphs. A process graph is a directed graph with labeled edges, in which the nodes represent

Use case: Heat Cooking Tank

Description: Heat a cooking tank to the temperature prescribed by the recipe of the juice to be mixed, and keep it at that temperature for the time prescribed by the recipe.

Reads: Cooking tank, Batch, Recipe.

Changes:

In: Operator: Batch ID.
Thermometer: Current temperature.

Out: Thermometer: Temperature request.
Heater: switch on, switch off.
Operator: heating finished.

Assumes: Juice is present in tank.

Results: Juice has been kept at desired temperature for the desired time.

Transactions:

- The system switched on the cooking tank heater.
- The system checks the cooking tank thermometer every 10 seconds. When the desired temperature is reached, the heater is switched off, when the temperature is too low, the heater is switched on again.
- When the end temperature is reached, the heater is switched off and a message is sent to the operator.

Figure 20. Schema for the Heat Cooking Tank use case.

states and the edges represent state transitions. Usually, there is an initial state, represented by a node that is pointed at by a small arrow. Figure 21 shows a process graph that represents a machine with two states and four transitions. The machine is nondeterministic because the start_heating transition may or may not lead to another state.

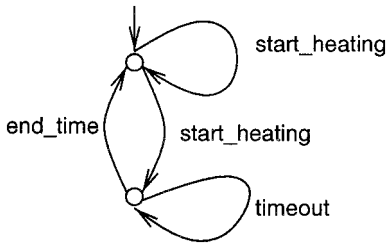

Figure 21. A process graph.

Process graphs may have infinitely many nodes and from any node, infinitely many edges may depart. They are used as interpretation structures for formal specifications in process algebra and dynamic logic [Baeten and Weijland 1990; Harel 1979]. Infinite graphs are not useful as drawing techniques and process graphs are not used in any of the methods reviewed in this survey. However, they are useful as a stepping stone for the discussion of the finite-state specification techniques that follow.

3.5.2 JSD Process Structure Diagrams. Process structure diagrams (PSDs) are used in Jackson Structured Programming (JSP) to represent the structure of files and of regular programs [Jackson 1975], and in JSD to represent the behavior of a system in a modular way (Appendix A.6). A PSD is a visual way to represent a regular expression by means of a tree diagram. For example, Figure 22 represents the same finite-state process as Figure 21. The leaves of the tree represent atomic actions; other nodes represent nonatomic processes. Left-to-right order-

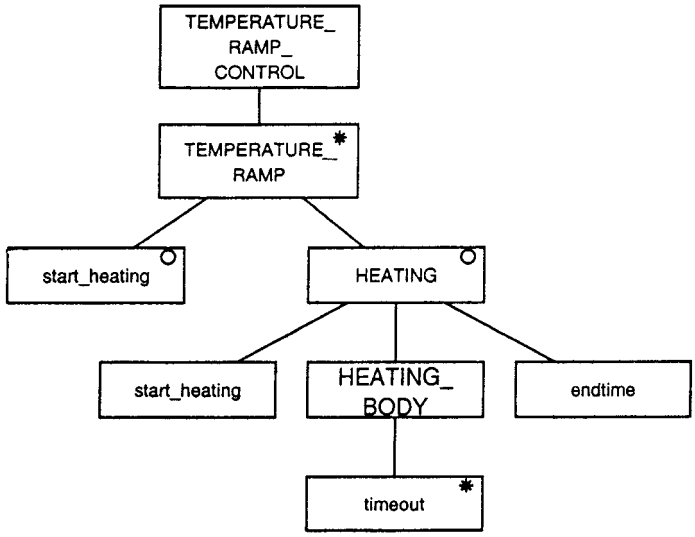

Figure 22. A process structure diagram.

ing of nodes represents sequence, except if the nodes are labeled with an "o", in which case the nodes represent alternatives. Iteration is specified by marking a box with an asterisk. PSDs are used in JSD to represent the lifecycle of entities as well as to specify functions of the system. For example, each process in the SND of Figure 11 would be specified by a PSD.

3.5.3 Finite State Diagrams. Process graphs assume that the most interesting thing about a behavior is the transitions, and they do not make any assumption about the number of states in a system. State transition diagrams (STDs) assume that the states are at least as interesting as the transitions and contain labels for states as well as for transitions. Finite STDs contain only finitely many states and transitions and this makes it possible, at least in principle, to draw finite STDs on a finite piece of paper. If the number of states is very small, this can actually be done in practice. Figure 23 shows an STD. The initial state is pointed at by a small arrow. We now know that the system starts in the IDLE state, a start_heating transition may lead the system to the CONTROLLING state, a timeout transition causes it to iterate over the CONTROLLING state, and an end_time transition causes it to return to the IDLE state.

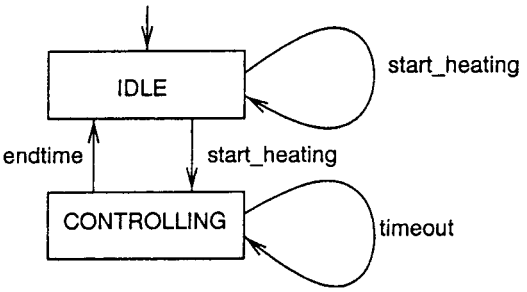

Figure 23. A finite state transition diagram with labeled states.

3.5.4 Extended Finite State Diagrams. The number of states that can be shown in one STD is usually extremely small compared to the number of states in which a system can be. The number of states can be increased by introducing variables that may be tested and updated by the finite state machine. The state of the system now is represented by the state of the STD plus the value of

the variables. We speak of the *global state* of the system that consists of the *explicit state,* represented by the nodes of the STD, and the *extended state,* represented by the values of the variables.

There are two ways to add variables to an STD: by means of local variables or by means of external variables. A local variable is declared together with the specification of the STD. A method employing this technique must define scope rules for these variables. Usually, the scope of the variable is the entire state machine specification, which means that any transition can contain tests and updates of these variables. Some methods allow variables whose scope is a single transition, where they are used as temporary variables to compute a result. Some methods also allow variables whose scope is a single state, where they are used to hold a result that can be used in the next possible transition. In all these cases, the explicit state represented by a node in the diagram stands for a set of global states, one for each possible value of the local state variables. Also, in all these cases, the computations that access and change the variables are specified as part of the state machine specification.

The second way to add variables to an STD is by means of external variables. An external variable is declared outside the specification of the STD but can be accessed by means of special operations that act as an interface between the specification and the variables. Computations that access or update the variables are now external to the state machine. For example, in dataflow models, data stores are external variables with respect to the control processes in the DFD. This means that a global state change now requires communication between the state machine and the external data stores. For example, in the DFD of Figure 7 the dashed circle represents a control process that must be specified as a finite state machine. The dashed arrows between the dashed circle and the other diagram components represent the communications required to synchronize a transition of this state machine and changes in the extended state.

Generally, object-oriented methods tend to choose the first option. A state machine is associated with an object, whose local variables (often called instance variables or attributes of the object) act as local variables of the state machine. SDL also chooses the first option. With some stretch of the imagination, JSD can also be interpreted as choosing the first option, because each PSD belongs to an entity, which can have local variables (called attributes) that may be changed by the actions in the PSD. Structured methods that use dataflow modeling choose the second option. Statemate uses both options, because statecharts allow local variables but activity charts allow data stores and regular activities to manipulate data.

The presence of variables (local or external) allows us to refine the specification of state changes in a number of ways. We list these ways here and give examples in Figures 27 to 29.

—A state transition may change the values of variables. We saw in Section 3.4.3 that changes can be specified in a declarative or imperative way. If the emphasis is on executable specifications, such as for SDL and Statemate, then imperative specifications are chosen.

—For each transition, a guard may be specified that says when the transition can occur. There are two interpretations of this. In a weak interpretation, the transition cannot occur if the guard is false. This means that if the transition occurred, then we know that the guard was true. However, if the guard is true, we do not know whether the transition can occur. The guard is in this case a necessary condition for the possibility of occurrence of the transition. In a strong interpretation, the transition can occur if and only if the guard is true. The guard is in this case a necessary and sufficient condition for the possibility of occur-

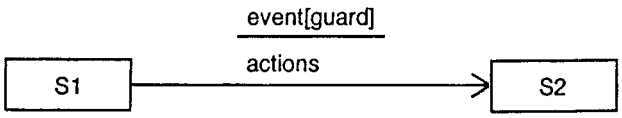

Figure 24. STD of a Mealy machine.

rence of the transition. During the process of building up a specification, we usually specify guards with the weak semantics until we know that we have specified all conditions for a transition, at which time we interpret the conjunction of all weak guards as a strong guard. One way of specifying a guard is to say that the guard is the conjunction of all preconditions specified for a transition. A guard for a transition may however be stronger than this conjunction, in which case some precondition is true of some states in which the guard prevents the transition from occurring. It is not desirable for the guard to be weaker than the conjunction of preconditions, because in that case there would be states in which the guard allows the transition but no precondition applies.

—The presence of variables allows us to include tests in a state machine that are used to determine what the next state will be. The next state is then determined by the value of the variables. Tests can be used to resolve nondeterminism such as that of the start_heating transition in Figure 23. The difference between a test and a guard is that a test determines which of a set of possible transitions will occur. A test consists of a guard for each of the possible transitions.

3.5.5 Mealy Machines. State machines never operate in isolation but must communicate with their environment. This is done by distinguishing *input events,* which are received from the environment, from *output actions,* which are sent to the environment. In all conventions, input events are associated with a transition, meaning that an input event occurrence triggers the transition, provided that the transition guard does not prevent this. The conventions for the definition of output actions differ. In a Mealy machine, output actions are associated with transitions. This is shown in the STD by separating the input events from the output actions of a transition by a horizontal line or by a slash (Figure 24). We call an STD of a Mealy machine a Mealy STD. If there are variables and we wish to define a guard for a transition, this may be added between square brackets behind the action. The actions may be updates of variables or output actions to the environment of the machine.

Figure 25 shows a Mealy state diagram with a decision state, following the YSM convention. The diagram refines the state diagram of Figure 23. It specifies the control process of Figure 7. The machine is initialized in the IDLE state and, upon reception of the event start_heating, generates output actions, one that triggers the data process Compare with desired temperature and one that enables a continuing data process that monitors the deadline when the heating will be finished. The machine then enters the state COMPARING, in which it awaits the outcome of the comparison with the desired temperature. This is called a *decision state.* These states are necessary because in this example, the machine cannot do any computations itself but must request external data processes to perform computations. The outcome of the comparison is either desired temperature reached or desired temperature not reached. These trigger a transition to the CONTROLLING state, turning the heater on or off according to the outcome of the test. The machine then periodically receives a timeout at which it

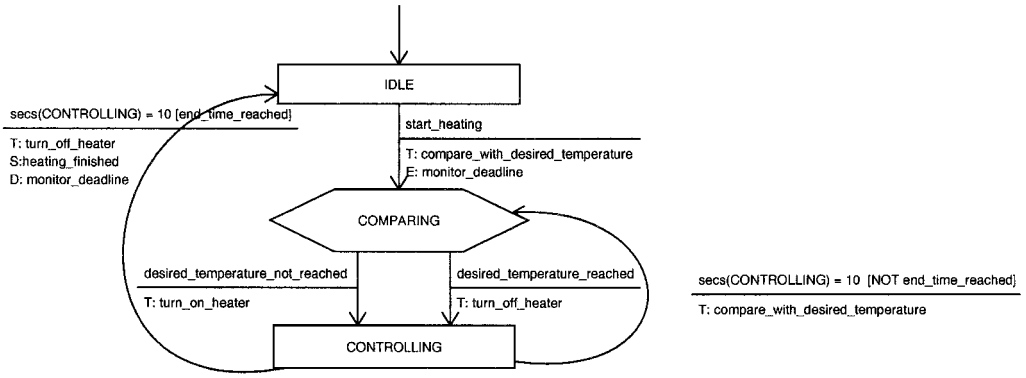

Figure 25. A Mealy machine with a decision state.

checks whether the desired temperature is reached and turns the heater on or off accordingly.

In YSM, there are actions to create and set named clocks. In addition, each state of a state machine has a clock that counts the time that has elapsed since the machine last entered that state. The event secs(CONTROLLING)=10 is a temporal event generated by the state clock of CONTROLLING. It occurs 10 seconds after the most recent entry in the CONTROLLING state. When this occurs, the machine checks whether the end time has been reached. If so, the machine stops monitoring the deadline, turns off the heater, outputs a signal that heating is finished, and returns to the IDLE state. If the end time has not been reached, another cycle of monitoring is initiated. The condition end_time_reached is a continuous event flow used in a guard for two transitions of the Mealy machine.

If a Mealy machine has many states or must react to many events, it may be impractical to draw an STD for it. One can then draw a state transition table. There are various layouts for such a table but a commonly used technique is to set off events against states and show in the entries which actions are generated and what the next state is [Hatley and Pirbhai 1987, pp. 82–83].

3.5.6 Moore Machines. In a Moore machine, outputs are associated with states. The meaning is that actions are performed upon entry of a state. This means that all transitions entering a state will generate the same output. In general, for every Mealy machine there is a Moore machine that has the same input-output behavior and vice versa [Hopcroft and Ullman 1979, p. 44]. Figure 26 shows an STD of a Moore machine that does the same as the machine of Figure 25. We introduced a state for every output produced. Note the output action finish controlling in state TURNING OFF, which triggers the transition to the FINISHED state. Moore machines are used in the Shlaer–Mellor method for the specification of object behavior (Appendix B.1).

3.5.7 Statecharts. A *statechart* is higraph without intersection but with Cartesian products, in which the nodes represent states and the directed hyperedges represent state transitions.[2] Node inclusion allows us to partition a state into substates. Cartesian products allow us to specify parallelism. In addition to these features, actions can be specified along transitions (Mealy), upon entry of states (Moore), and exit of states. Statecharts also use local variables to represent an extended state.

[2] There is a version of statecharts with intersection, but this is not part of the Statemate language discussed in this survey [Harel and Kahana 1992].

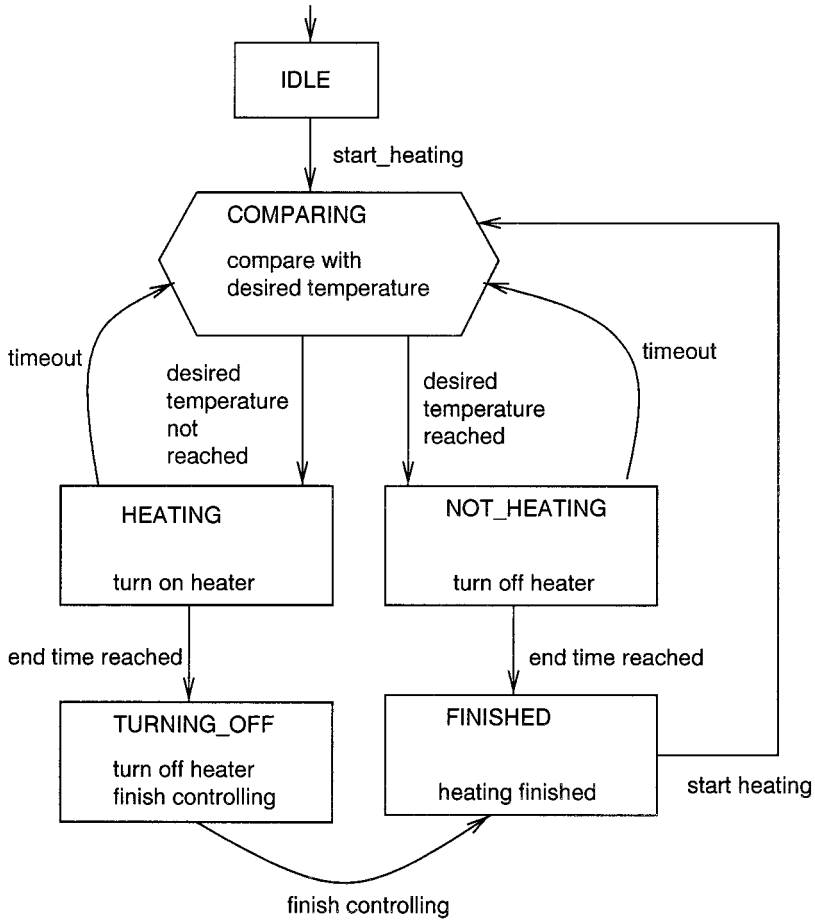

Figure 26. STD of a Moore machine with a decision state.

Figure 27 shows a statechart for the juice plant controller. In Statemate, a statechart corresponds to a control activity in an activity chart, just as in YSM a Mealy machine corresponds to a control process in a DFD. The MONITOR_TEMPERATURE state is partitioned into two or-states, HEATER_ON and HEATER_OFF. We assume that the comparison of the current with the desired temperature is done externally by an activity in the activity chart, called Compare_with_desired_temperature, just as in the DFD of Figure 7. The start_heating event causes a transition into MONITOR_TEMPERATURE, whose initial state is COMPARING, as indicated by a small arrow. Upon entry of this state, the machine starts the activity compare_with_desired_temperature. The answer sent back by this activity determines whether the heater is turned on or off. Note that these actions could also have been associated with the HEATER_ON and HEATER_OFF states. The timeout action takes the machine out of any substate of MONITOR_TEMPERATURE. The variable end-time must be made available to the statechart as an input data flow in the corresponding activity chart, in which this statechart corresponds to a control activity.

It is possible that the timeout leaving MONITORING_TEMPERATURE occurs at the same time as a timeout leaving HEATER_ON or HEATER_OFF. There are

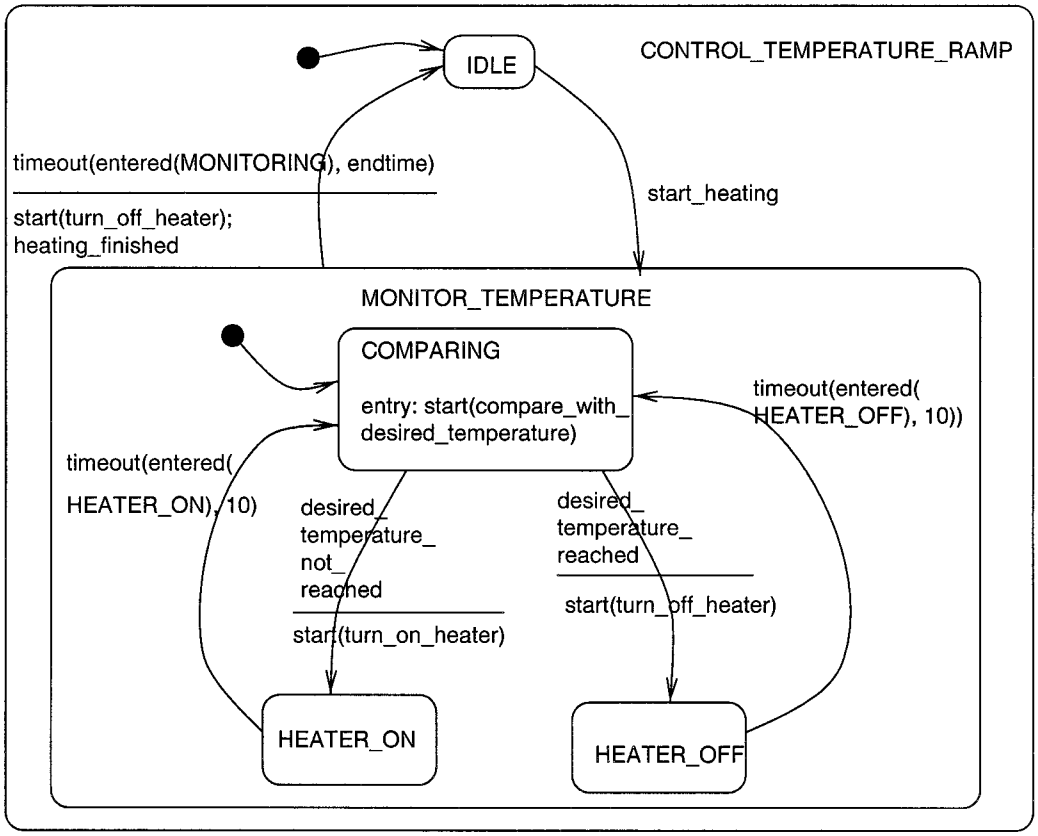

Figure 27. A statechart.

different possible semantics to resolve this conflict. The current Statemate semantics gives priority to the timeout leaving the superstate.

Note that monitoring the end time is done by the statechart itself. This is contrasted with the DFD in Figure 7, where the activity Monitor deadline is external to the control process Control temperature ramp.

The comparison between the current and the desired temperature is done by an activity external to the statechart. This is analogous to the DFD of Figure 7. Because statecharts allow the declaration and manipulation of local variables, a different solution is possible, in which the current and desired temperatures are made available to the control activity through input flows and the choice to turn the heater on or off is made by means of a test.

If the control activity must perform other jobs in parallel to monitoring the temperature, then this can be specified by adding parallel components. For example, Figure 28 added a MONITOR_PRESSURE process in parallel to the temperature monitoring process of Figure 27. This parallel composition cannot be specified conveniently in one Mealy or Moore machine. In YSM, the DFD would have to specify a separate control process, with its own Mealy machine, corresponding to the MONITOR_PRESSURE process in Figure 28.

An action generated by a state transition is broadcast to all parallel components in the same statechart. This may trigger other transitions, which in turn

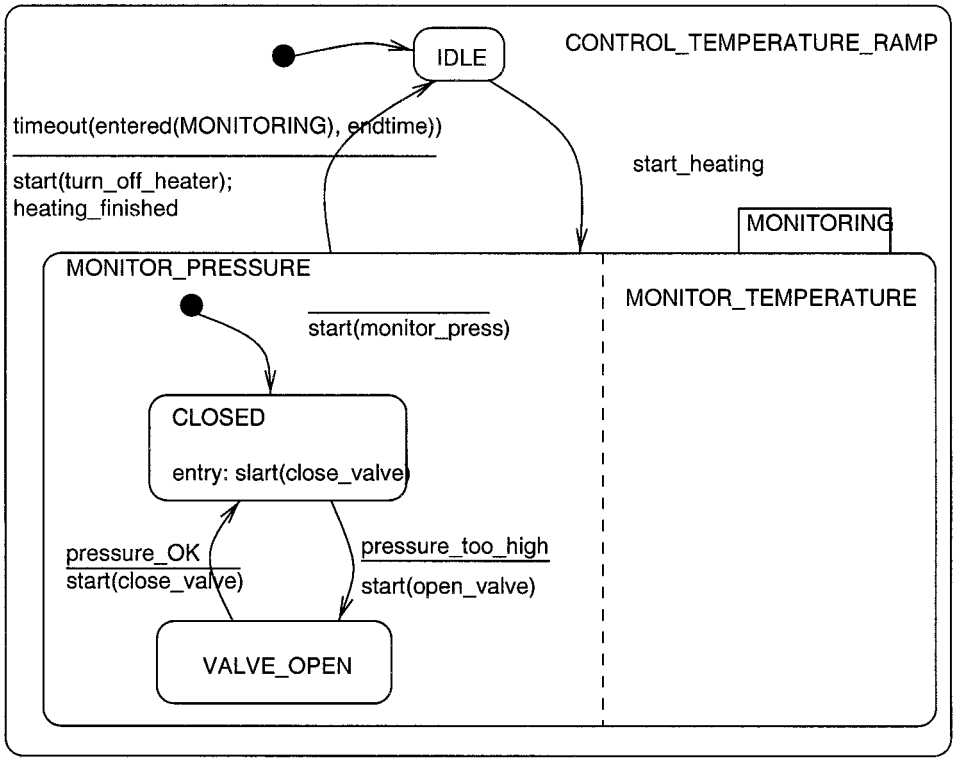

Figure 28. A statechart with parallel substates.

may generate other actions. This process comes to a halt when no more transitions are triggered.

Statemate has defined a large number of special events, actions, and conditions that can be used in statecharts. An *event* dealing with real time is timeout(e, t), which occurs t time units after the most recent occurrence of event e. An *action* dealing with real time is schedule(e, t), which schedules the event e to occur t time units after now.

Formal semantics of statecharts are defined by Harel et al. [1987] and Pnueli and Shalev [1991]. Beeck [1994] lists over 20 different formal semantics of statecharts. The semantics used by Statemate is described in an internal i-Logix report [i-Logix 1991b]. A summary of this is presented by Harel and Naamad [1996].

3.5.8 SDL State Diagrams. SDL state diagrams are extended finite state diagrams with input and output and timers. They are used in SDL to specify the behavior of processes in the leaf blocks of a block diagram. Figure 29 shows an SDL state diagram that declares two local variables and two timers. A timer is an object that is owned by the state machine in whose specification it is declared, and that can send timeout signals to the machine. The type Recipe must have been defined in an enclosing block as a record type. The machine starts in the IDLE state. When it receives the input signal start_heating, it initializes the variable desired_temperature by accessing a field of recipe and sets the two timers. These inputs must be input signals of the process in the corresponding block diagram. It then tests the current temperature and sends out a signal to turn the heater on or off. The test is called a decision in SDL, represented by a dia-

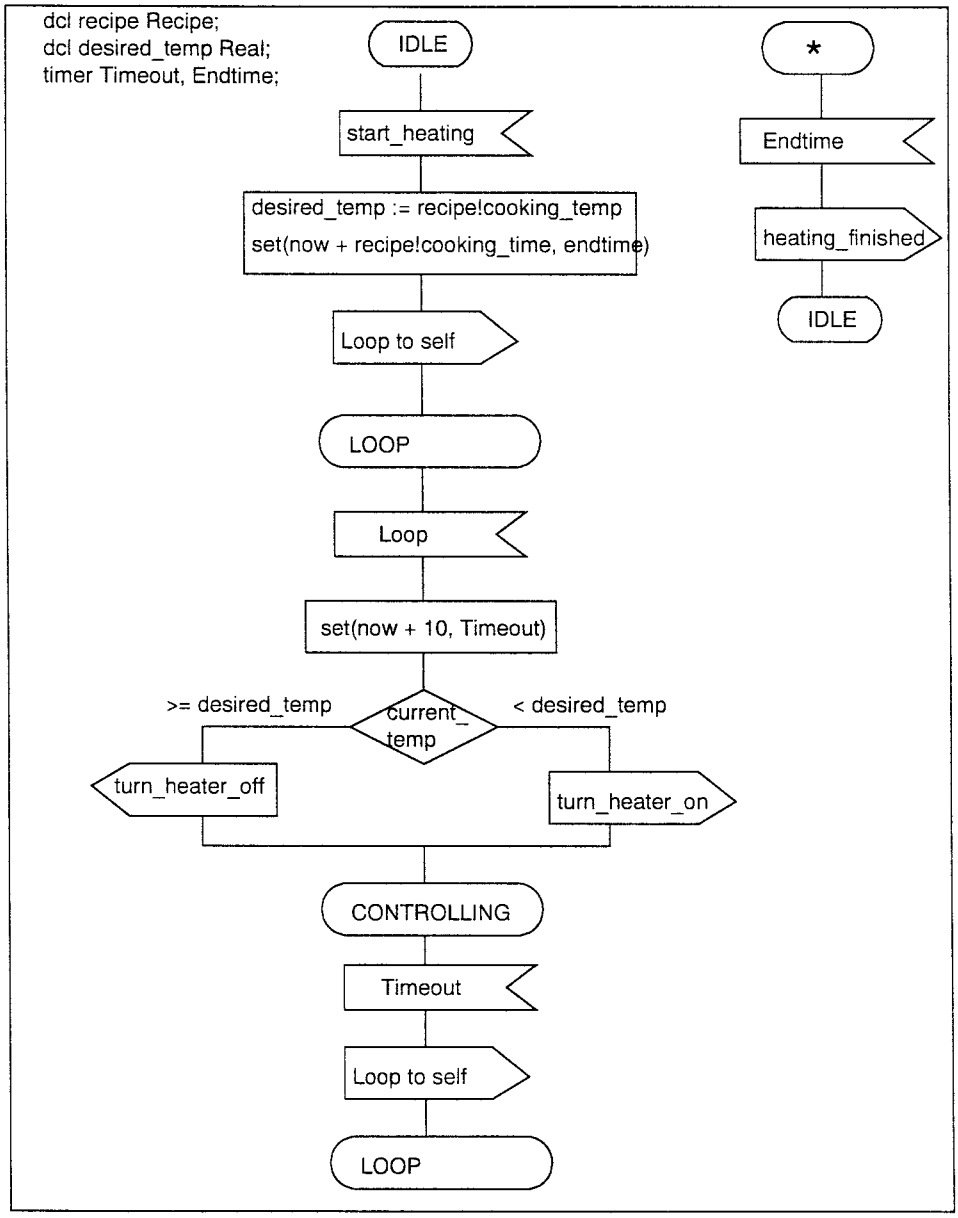

Figure 29. An SDL state diagram.

mond. The machine then periodically receives a timeout signal, at which it checks the current temperature and sets a new timeout. An assignment to variables or an update of a timer is done in an SDL task, represented by a rectangle. States are represented by rectangles with rounded sides and inputs and outputs by flag-like symbols, pointing inward or outward. The asterisk state in Figure 29 refers to any state in the current diagram. It is used to specify that upon reception of a signal of the Endtime timer, it turns off the heater, sends out a signal that heating is finished, and returns to the IDLE state.

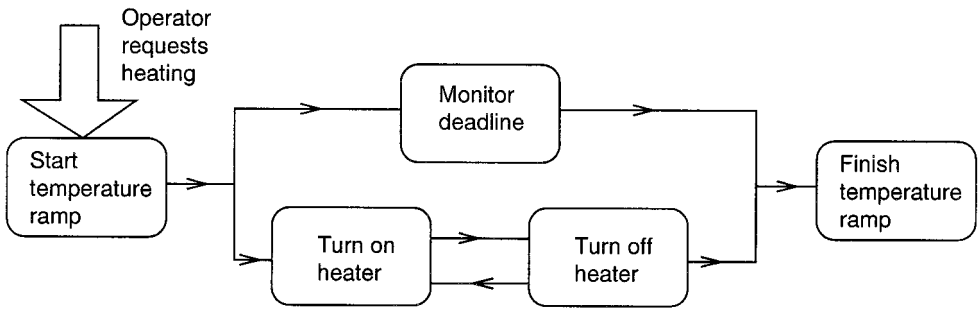

Figure 30. A simple process dependency diagram.

Different processes in a block diagram communicate through signal routes (Section 3.3.7). An alternative means of communication between processes is by exporting a variable from one process and importing it to another. By executing an export statement, the exporting process makes the value of the variable available to any process that imports the variable. When an importing process executes an import statement, this reads the value that the variable had at the most recently executed export statement. SDL diagrams have a formal semantics, defined as an ITU standard [ITU 1992].

3.5.9 Process Dependency Diagrams. Process dependency diagrams are directed hypergraphs in which the nodes represent ongoing activities, called processes, and the edges represent transitions between these activities. They are used in information engineering (Appendix A.5) to represent precedence relations between activities. Figure 30 shows a simple process dependency diagram. The event that triggers the processes in the diagram is represented by a large arrow. The hyperedge leaving start temperature ramp represents parallel execution of the target processes monitor deadline and turn on heater. There is also a convention for alternative execution of processes. Process dependency diagrams allow the representation of cardinality properties of precedence relationships. Figure 31 shows that the process produce quality report is preceded by one or more test batch quality processes, using the traditional crow's foot notation to represent the cardinality *one or more.*

Figure 31. A process dependency diagram with cardinality.

A process dependency diagram is similar to a dataflow diagram in that both diagrams represent a collection of related processes. The difference is that an edge in a process dependency diagram represents logical precedence and an edge in a dataflow diagram represents a dataflow. Because of this, process dependency diagrams are closer to state transition diagrams, in which edges represent the transition from one activity to another activity. Process dependency diagrams can represent triggering events, parallelism, choice, and the cardinality of process connections, none of which can be present in a dataflow diagram. However, process dependency diagrams can be extended with data stores, giving them the appearance of dataflow diagrams [Martin 1989b, p. 271].

Related conventions are used in the Martin–Odell method (Appendix B.6) and in the UML (Appendix B.19). The Martin–Odell method uses event diagrams which are similar to process dependency diagrams but without event

arrows and without the explicit representation of cardinalities [Martin and Odell 1995, pp. 118 ff.]. The exact nature of a dependency of one process upon previous processes can be specified in an event diagram by a control condition. For example, a control condition can specify that a process can be performed if at least one of the immediately preceding processes is finished, or if all of them are finished.

In the UML, *activity diagrams,* not to be confused with SADT activity diagrams (Section 3.3.3), are defined as statecharts all of whose states represent the performance of activities and whose transitions are triggered when an activity is terminated. A similar technique is defined in Mainstream Objects (Appendix B.15). The result is very similar to process dependency diagrams without cardinalities or events [Rational 1997a]. Activity diagrams also allow the representation of input and output objects of an activity.

3.5.10 Summary. There is considerable similarity between the different notations reviewed in this section. All notations represent states by labeled nodes and transitions by labeled directed (hyper)edges. The edge labels represent the input event that triggers the transition, the output actions generated, and possibly the actions performed on local variables. All techniques except process dependency diagrams have a formal semantics. There is exactly one semantics of SDL state diagrams laid down in the standard [ITU 1992], but there are many different possible semantics of statecharts [Beeck 1994]. Statecharts are very popular in object-oriented methods but the varieties of statechart-like notations defined in those methods do not all have a precisely defined semantics.

Process dependency diagrams hardly have any semantics defined for them. For example, cardinality constraints between processes require a theory of equality between processes. The statement that one process of type P receives messages from many processes of type P′ is only meaningful in the presence of a theory that says when processes are counted equal and when they are counted as different. This is the province of process algebra [Milner 1980; Baeten and Weijland 1990], which is far removed from the realm of process dependency diagrams. Martin–Odell event diagrams and UML activity diagrams likewise contain constructs that, at the time of this writing, are nowhere defined precisely. This means that, say, different case tool vendors can implement different execution algorithms for these diagrams and both claim to implement the "right" semantics.

We skipped over the details of the semantics of time in state machines. For example, one may choose a point or interval semantics of time, use a discrete, dense, or continuous model of time, and assume that transitions take time or are instantaneous. Each of these choices leads to important differences in the behavior specified by a state diagram. The semantics of time is carefully defined in SDL and in the Statemate semantics of statecharts.

Related to this is the variety of communication semantics that one can use. We saw already in Section 3.3.10 that some methods are not very explicit about the semantics of communication diagrams. The set of choices that can be made increases if we realize that we must specify whether input events are queued upon reception, what happens if an event cannot trigger a transition, and so on. Any method that employs these notations must make these semantic choices clear. Again, these issues are explicitly dealt with in SDL and the Statemate semantics of statecharts.

4. ANALYSIS AND COMPARISON

Appendices A and B review how 6 structured and 19 object-oriented methods use the techniques described in the previous section. Tables A.I and A.II of those appendices give a quick survey of

techniques used in the reviewed methods. In this section, we summarize our findings. We start with the way in which external interactions and conceptual decomposition are specified in structured and object-oriented methods. Next, we look at the way behavior and communication are specified in each type of method. Finally, we take a brief look at the complexity reduction techniques used in the reviewed methods.

4.1 External Interaction Specification

The three relevant aspects of external system interactions are external functions, communication and behavior. Looking at the first three columns of Tables A.I and A.II we see that most of the techniques employed by different methods to specify external interactions can be freely combined. Structured techniques use event-response pairs to specify system transactions (YSM) and a function refinement tree that relates the transactions to the system mission (IE). Event-response pairs can be specified declaratively by means of pre- and postcondition pairs or imperatively by means of executable programs. A statement of purpose summarizes the functionality of the system by listing the responsibilities of the system as well as the functions that the system will not have. At a greater level of detail, external communication and behavior can be illustrated with sequence diagrams (SDL), and the details of the dataflows between the system and its environment can be specified by a context diagram.

Object-oriented methods have little to add to this. We noted already that use case diagrams can be viewed as partial context diagrams that show which external entities communicate with the system during particular functions (use cases). The other addition of object-oriented methods is the use of collaboration diagrams as an alternative to sequence diagrams to illustrate external behavior and communication of the system. This pretty much exhausts the techniques used in the 25 reviewed methods for the specification of external interactions.

4.2 Decomposition Specification

4.2.1 Kinds of Components. The essential differences between structured and object-oriented methods exist in the way they conceptually decompose the software product. Structured analysis uses DFDs to specify system functions. When a DFD contains dataflows between data processes then it actually shows a conceptual decomposition in which several data processes interact to realize an external function. When it contains control processes, the DFD contains event flows that show how the control processes cooperate with other processes to realize external system functions. In both cases, what is shown is not a collection of external system functions but a collection of communicating processes that work together to realize external system functions. This is why we listed DFDs as a technique to specify a conceptual decomposition. This view of DFDs is even more justified if the DFD is hierarchical, for the DFD hierarchy then actually shows a conceptual decomposition hierarchy.

Suppose we flatten a DFD hierarchy and reorganize its data processes so that each data process corresponds to one atomic external system transaction. We ignore control processes for the moment. Then there will be no interfaces between the data processes, because different data processes correspond to different system transactions. Such a DFD is partitioned according to the event partitioning principles introduced by McMenamin and Palmer [1984]. It shows only what the external transactions are and what their interfaces to external entities and system memory (data stores) are. Such a diagram still shows a decomposition, that is, a decomposition of the system into data processes and data stores. Basically, this is the separation between memory and processor typical of von Neumann com-

puters. We call this decomposition principle *data process separation.* In such a decomposition, there are two kinds of components: data stores that have memory but no activity and data processes that have activity and no memory. The decomposition principle of DFDs is actually a bit more complex, because only instantaneous data processes have no memory. Continuing data processes may have memory. Nevertheless, data process separation is one of the major decomposition principles imposed by DFDs.

Data process separation is opposite to the *update encapsulation,* in which instantaneous data processes are modeled by means of updates that are encapsulated with the state that is updated. It is true that YSM actually uses the concept of abstract data type, which encapsulates the specification of operations with the operated upon data. Example abstract data types are integers, characters, and parameterized types such as lists and sets of items. However, YSM does not extend this idea to the encapsulation of updates with the state that is updated. We can simulate objects in a DFD by a data store that contains the state of all instances of an object class, and a number of instantaneous data processes corresponding to the object updates that are the only processes to access the data store. This simulation is also a vivid illustration of the difference between data process separation and update encapsulation.

Note as an aside that update encapsulation implies the need for a concept like object identity as that which remains invariant under all possible changes of state. Without such a concept, the concept of state change would be meaningless.

Dataflow models are also characterized by a second separation, that of data processes from control processes. We call this the principle of *control process separation.* This separation too is abolished in an object-oriented decomposition, where a state machine can be specified for each object class, with the meaning that each instance of that class executes an instance of that state machine. Data process separation and control process separation jointly take care of the idea to use external variables for extended finite state machines. The data stores act as external variables for the state machines, and data processes as external updates of these variables.

Putting all of this together, we get that DFDs use four kinds of components; data stores, instantaneous data processes, continuing data processes, and control processes. Object-oriented methods use only one kind of component, the object, that encapsulates data (local state), updates (local operations), and control (local state machine).

Turning to the other structured methods, we see that information engineering uses DFDs and hence follows the same decomposition principles as YSM. SADT only recognizes activities as conceptual components and hence does not apply the principle of data process separation or control separation. Statemate contains the principles of data process separation and control separation, but it in addition allows the encapsulation of a local state with the control processes. If we drop activity charts from Statemate models, all data processing must be encapsulated in statecharts and we get a model that is very similar to object-oriented models. SDL does not contain the principle of data process separation or control separation. It does not contain separate data stores or data processes at all. Note that although SDL state machines encapsulate updates, these are not named but are "programmed" as tasks. This differs from object-oriented techniques that allow external access to object operations by their name only. JSD too does not contain a separate data store construct nor does it distinguish data processes from control processes. Updates are called actions and specified separately from PSDs.

We conclude that data process separation and control process separation are typical for DFDs only and do not occur

in structured methods that do not use DFDs. Moreover, Statemate, SDL, and JSD all encapsulate updates in their state machines, which puts them all close to object-oriented methods. Because Statemate and JSD do not support the concepts of class or inheritance, we could call them object-based rather than object-oriented [Wegner 1992]. SDL 92 has moved closer to object-orientation because it includes the concept of inheritance [Møller-Pedersen et al. 1987; Færgemand and Olsen 1994].

4.2.2 The Subject Domain. The decomposition of system memory is represented in YSM in two ways, by a decomposition into data stores and by a decomposition into entities and relationships, specified by an ERD. In object-oriented methods the conceptual decomposition into objects is always represented by a class diagram-like model, which is a close relative of the ERD. Now, what do these diagrams represent exactly? In Section 3.2.1 it is remarked that one cannot tell by looking at an ERD whether it represents a decomposition of the environment of the system or a conceptual decomposition of the system itself. This ambiguity is related to a fundamental characteristic feature of software, which we now make explicit.

A software system is a system that manipulates and stores data. The defining characteristic of data items is that they refer to something. A data item is a symbol that has a meaning which is not given by nature but laid down in a convention usually called a dictionary. We call the part of the world to which the data refers the *subject domain* of the data. Another frequently used term is *universe of discourse* (UoD), but we stick to the term subject domain here. The subject domain always lies outside the software being specified, but it may lie outside or inside the computer. There are exceptional cases where we can define a symbol that refers to itself, but for software specification these cases are not interesting. For example, the subject domain of a personnel database system consists of the relevant aspects of the employees of a company, such as name, address, birth date, salary, and the like, and the subject domain of a juice plant control system consists of the relevant parts of the juice plant, such as heating and storage tanks, heaters, temperature sensors, and the like. The subject domain of a database dictionary, however, is a database system, which resides inside a computer. The subject domain of a software product may even be an abstract conceptual world that does not exist in the physical world. For example, the subject domain of a graph editor consists of the set of possible graphs that can be drawn.

The subject domain plays an important role in the decomposition principles used in object-oriented methods, but the distinction between the subject domain and the software product being specified is not made clearly in most methods. Exceptions are JSD (Appendix A.6), which is the first method in which the distinction is made explicit, and Syntropy (Appendix B.14).

We noted that the meaning of a data item is not given by nature but by a convention according to which people agree that a certain observable symbol occurrence stands for something else. If we specify a software system, we must include this meaning convention in our specification. This means that we must include a dictionary in the software specification that defines the meaning of data items. The data dictionary used in dataflow modeling contains such information, but it also contains additional information about data types and representation formats of data, which have no meaning in the subject domain. To make clear that we do not have this kind of data dictionary in mind, we call the meaning definition of the data items a conceptual model. Thus, a *conceptual model* of a software product is a definition of the meaning of the data items in terms of the subject domain.

In database modeling, the conceptual

model always takes the form of an ERD and then represents a decomposition of the subject domain. This ERD specifies the meaning of the data by simply specifying the kinds of things (entities and relationships) to which the data can refer. For every entity or relationship in the subject domain there will be a corresponding surrogate in the database [Hall et al. 1976]. As a consequence, the ERD represents a decomposition of the subject domain as well as the conceptual structure of the data in the database. Note that the ERD can be used to represent an aggregation hierarchy of entities and components in the subject domain, but that the surrogates that represent these entities all reside at one level of the aggregation hierarchy of the software product.

In the specification of control-intensive software systems, the subject domain consists of the entities whose behavior is to be registered and/or controlled. In this case, a decomposition specification of the subject domain does not necessarily correspond with a conceptual decomposition of the software system, because the system does more than store and remember all events that occur in the life of the subject domain entities. However, as shown in the following section, one can use a decomposition criterion that produces an isomorphy between the subject domain and at least part of the conceptual software components. Here we note that even if there is no such isomorphy, the meaning of the data manipulations performed by the system must be defined in terms of actual and desired behavior of entities in the subject domain. We always need a conceptual model of the subject domain to define this meaning.

Note that since in object-oriented specifications, operations are encapsulated with the data they operate on, a class diagram not only represents the meaning of the data items but also the meaning of data manipulations in terms of the subject domain. For example, JSD makes a model of the behavior of the entities in the subject domain before making a model of software objects.

In structured methods of the Yourdon school, ERDs are used to represent the conceptual structure of the data manipulated by the system. The functions of the system are represented by a DFD [Flavin 1981; Yourdon 1989; Yourdon Inc. 1993]. In other structured methods, ERDs are not used. These methods are oriented towards systems that are not data-intensive. In object-oriented methods, class diagrams are used to represent a conceptual decomposition of the software where the decomposition principle is that objects in the subject domain should correspond to conceptual objects in the software. In JSD and Syntropy, the distinction between a subject domain decomposition and a software specification is made fully explicit: the decomposition of the subject domain is modeled separately from the decomposition of the software product, and the link between the two is made explicit (Appendices A.6 and B.14).

4.2.3 Decomposition Criteria. Structured and object-oriented methods seem to use quite different decomposition criteria for software. A closer look reveals that this is not the case. We review a number of decomposition criteria that are used in structured and object-oriented methods.

—In *functional decomposition,* conceptual components correspond to system functions [DeMarco 1978]. In this kind of decomposition, we choose a level in the function refinement tree and decide that the functions at that level are to be specified as conceptual components. We thus map the horizontal dimension of a function decomposition table to the vertical dimension. The result is that the allocation entries in the table appear on the diagonal.

—In *event partitioning,* each conceptual component corresponds to the processing between an event and the response by the system [McMenamin

and Palmer 1984]. This is a kind of functional decomposition, but now we choose the lowest level in the function refinement tree, that of atomic system transactions where each transaction consists of a conceptually atomic event-response pair. There will be one conceptual component for each external system transaction.

—In *device-oriented decomposition,* one conceptual component is defined for each external device to be controlled; the task of the component is to control the device [Yourdon Inc. 1993, pp. 329, 355, 509]. Since each device can generate more than one event to which the system must respond, this decomposition is at a higher level of granularity than event partitioning. Unlike functional decomposition, it does not group event-response pairs into more abstract functions, but it groups them according to the device that originates the event.

—In *subject domain-oriented decomposition,* one conceptual component is defined for each external entity about which data is to be stored or whose behavior must be controlled [Jackson 1983]. This differs from device-oriented decomposition, because a device connected to the software system may not itself be the entity whose behavior is to be controlled, but merely an intermediary that lets the system and the subject domain entity communicate with each other. Since one entity may be observed by the system through several devices, this may lead to a decomposition at a higher level of granularity. It is a decomposition that in any case stays close to the meaning of the data manipulations of the system.

A more detailed exposition of these decomposition criteria is given by Cook and Daniels [1994], Gomaa [1993], Jackson [1983], and Shumate and Keller [1992].

All of these decomposition criteria are compatible with object-oriented decomposition as well as with the separation of data stores, updates (data processes), and control characteristics of DFDs. Functional decomposition, event partitioning, and device-oriented partitioning all originate from structured analysis. The idea of subject domain-oriented partitioning was already present in Simula [Dahl and Nygaard 1966], which stands at the root of the tradition of structured programming. The idea led a sleeping existence until 15 years later when it became a central partitioning criterion in JSD, which too has a background in structured programming [Jackson 1975]. It is not difficult to structure a YSM model according to the principle of subject domain partitioning and many examples in the structured analysis literature actually use this partitioning without mentioning it. All ERDs given by Yourdon Inc. [1993] and Goldsmith [1993] use, for example, this principle. To be fair, it should be noted that they do not partition the data processes according to the same principle. Events that update the state of entities are not encapsulated in the specification of those entities.

Conversely, object-oriented decomposition is compatible with all of the decomposition criteria mentioned previously. It is perfectly possible to define software objects that correspond to transactions, or to devices, to system functions, or to subject domain entities. This is most explicit in Objectory (Appendix B.5), where entity objects correspond with subject domain entities, control objects with control functions, and interface objects with devices. It is also present in other object-oriented methods, but there the distinction between software objects that represent subject domain entities and software objects that embody software functions is blurred. For example, Figure 6 is based upon a class model given by Shlaer and Mellor [1992, p. 90]. It includes the object class Temperature Ramp, which is a control process in the DFD of Figure 7. The temperature ramp object really corresponds to the control process, for the state diagram specified by Shlaer and

Mellor is equivalent to the state diagram of the control process (Figure 25). Nevertheless, the distinction between software objects that correspond to subject domain entities and software objects that correspond to control functions is not made by Shlaer and Mellor. The confusion is even greater in the Fusion method (Appendix B.12), where the system is defined to be a part of the subject domain. Magically, the system contains software objects whose function is to represent subject domain entities or to act as an interface with those entities. These software objects cannot possibly be part of the domain. The problems in teaching Fusion, reported by Eckert [1996, p. 282], are caused by this confusion.

The compatibility of object-orientation with different decomposition heuristics should not be surprising: using objects means that we encapsulate a state and updates of the state into little systems called objects. Using a particular decomposition criterion means that we are guided in a certain way in choosing which objects we have in the system. The decision to use objects has nothing to do with the decision of which decomposition criterion to use.

We conclude that the use of a particular decomposition criterion is not the distinguishing feature of structured analysis versus object-oriented analysis. The distinguishing feature is the encapsulation of data, operations, and control into objects versus the separation of these elements in dataflow modeling. However, we repeat that this is a characteristic of DFDs. It follows that DFDs are incompatible with object-oriented techniques. We already saw that Statemate, SDL, and JSD are structured methods that do not use DFDs. They all use techniques that encapsulate data processing in control processes and access to the state of control processes is regulated by flows, signals, channels, data streams, and state vector connections.

4.3 Behavior Specification

For the specification of component behavior we have the same repertoire of techniques available as for the specification of external behavior. Generally, external behavior is defined implicitly as the result of a collection of communicating component behaviors. For example, DFDs and activity charts define external behavior by means of a collection of control processes, SDL defines the behavior of a system by means of a collection of communicating processes defined for leaf blocks, JSD defines system behavior by means of communicating surrogate processes and function processes, and all object-oriented methods define system behavior as the result of a collection of communicating objects. There are a few exceptions to this. SADT does not define behavior by means of state diagrams and some methods employ process dependency diagrams to represent behavior that do not localize behavior to single components. Fusion does not define component behavior at all in the analysis phase.

Statemate and almost all object-oriented methods use statecharts or one of their many variants to define object behavior. Statemate [i-Logix 1991b; Harel and Naamad 1996], Harel and Gery [1996], and Cook and Daniels [1994] give a precise semantics to the notation, including the behavior with respect to time. In all other cases, variants of the notation are introduced for which published work does not define a semantics. As a consequence, when different users of these notations interpret a diagram in different ways, there is no other way to discover the "true" meaning of the diagram than to ask the original author of the diagram—which defeats the purpose of using the diagram as a means of communication between different people. Of course, the diagrams can still be useful to record a half-baked idea for discussion, to be made more precise later.

The semantics of SDL state machines

is defined in terms of an underlying virtual machine that can execute SDL models [Belina et al. 1991]. Similarly, ROOM [Selic et al. 1994] gives a semantics to RoomCharts in terms of an underlying virtual machine that can execute ROOM models.

4.4 Communication Specification

Communication specification techniques can be used to specify the communication of the system with external entities or to specify communications between the components of the system. Structured methods generally view a system as a collection of communicating activities. They represent the communication between component activities by means of a graph in which the nodes represent activities and the edges possible communications. Object-oriented methods view a system as a collection of communicating objects. Behind a confusing difference in terminology, Table A.II shows an overwhelming agreement among object-oriented methods for the use of sequence diagrams and collaboration diagrams to represent sequences of messages between objects. The sequences specified by these diagrams must be consistent with the specification of object behavior.

Just as for behavior specification, there is a need for precision in the specification of communication. The issues include synchronous versus asynchronous communication, delayed versus immediate arrival, ordering of messages preserved or not, handshaking versus multicast or broadcast, channel capacity, queueing of incoming messages, and so on. This precision is provided by the same methods that define a precise behavior semantics. Other methods do not give a precise semantics of communication in their published work. Again, this defeats the purpose of using the notation for anything other than organizing half-baked ideas in a rough sketch, to be worked out later.

4.5 Complexity Reduction

Large models are simplified by introducing hierarchy. SADT activity diagrams, DFDs, Statemate activity charts, and SDL block diagrams all introduce hierarchy in more or less the same way information engineering simplifies large ERDs by partitioning them into subject areas, Shlaer and Mellor introduce domains, and in the UML, large class diagrams are simplified by parceling them into packages. An important issue is whether the hierarchy is a convenient way of organizing a large description or whether it represents a hierarchy of subsystems. SDL allows both interpretations [Belina et al. 1991, p. 11]; the other techniques view it as a convenient way of organizing a large description. Shlaer and Mellor, Syntropy, and Octopus introduce the concept of a subsystem as a way of partitioning a large system into simpler parts.

5. DISCUSSION AND CONCLUSIONS

This survey has shown that systems engineering offers a framework in which to integrate the techniques of structured and object-oriented software specification. The framework simply says that the system under development must be viewed as an aggregation hierarchy of subsystems, and is itself a system in such a hierarchy. To specify and design the system, we must specify desired external interactions, specify a decomposition, and allocate external interactions to component interactions. An issue that seems to divide structured from object-oriented design is the use of functional decomposition versus subject-domain decomposition. It is argued in Section 4 that this is not an essential division, as both heuristics can be used in object-oriented as well as structured methods. The essential division lies in the use of the DFD technique.

DFDs and their close relatives, activity charts, are incompatible with the use of object-oriented decomposition be-

cause of the presence of data process separation and control separation. The DFD technique forces the analyst to distinguish, at some level of the aggregation hierarchy, data stores from data processing components and at that level, we get a decomposition that is incompatible with the object-oriented philosophy. All other reviewed techniques are compatible with the structured as well as the object-oriented philosophy of software specification and, with the exception of SADT activity diagrams, we encounter the same techniques in structured and object-oriented methods. Note however that DFDs are also recommended by a number of object-oriented methods. This blurs the line between structured and object-oriented methods even on this point.

The technique most congenial to object-oriented decomposition is the extended finite state diagram, because this allows the specification of local state, state transitions, and behavior. We have seen that this technique is used in some structured and most object-oriented methods. Again, this blurs the dividing line between structured and object-oriented methods. The conclusion is that object-oriented methods have adopted structured techniques in a new guise. We can benefit from making this "technology transfer" explicit. One benefit is that this allows us to pinpoint the agreements and differences between structured and object-oriented methods, as we have done in this article. This should ease the transition from structured to object-oriented techniques or, better still, it should ease the eclectic use of techniques from both approaches.

Following up this last suggestion, how can one get the best of both worlds? First, it has been shown here that structured and object-oriented techniques for the specification of external system functionality can be easily combined. The statement of purpose, function refinement tree, event-response list, declarative specification of input-output relations, use case diagram, and context diagram all express some interesting property of the system. Also, the analysis in this article has at least made plausible that the properties expressible by these techniques are related in a simple manner and that it is feasible to define a set of coherence rules that link specifications of different views of these properties. Definition of these coherence rules requires a more precise semantics of the notations than is now available and is the subject of current research. It should be emphasized that even when such a precisely defined set of coherence rules is available, that this does not mean that in all cases we should use all of these techniques at the same time. However, it does mean that in each individual specification we can use any convenient combination of these techniques.

In the case of conceptual system decomposition, there is more variety. Nevertheless, there is an overwhelming agreement that the decomposition must be represented by a class diagram, component behavior by a statechart, and component communication by sequence or collaboration diagrams—more or less the UML way of specifying decomposition. However, the sheer number of methods that propose something like this structure indicates a problem: they are all different methods. What makes them different is that each method defines its own syntactic variation of some of the techniques, and gives its own interpretation to it. Some of the syntactic idiosyncracies are quite complex, and some of the methods only give a vague indication of the semantics of their techniques and other methods do not give a semantics at all. Nor are the methods very clear about the way in which their techniques must be connected into one coherent system specification.

The response of industry to this variety has been to start a standardization process [OMG 1996]. This led to the Unified Modeling Language (UML Version 1.1, see Appendix B.19), which consists of the original UML submission (Version 1.0) and elements of five other submissions. As standardization efforts

go, the result of standardization is often a compromise between parties with different interests and this is usually not the simplest possible solution.

The research issue that evolves from this is that current and future notations should be given a precise semantics. Each notation used in software specification should have an unambiguous declarative semantics that tells us what it means, and does not leave room for individual variations in meaning that exist only in the mind of the reader or writer. Whenever such variations exist, something is bound to go wrong in the communication from user to analyst to designer and programmer. A second reason for having a precise semantics is that the value of a specification is considerably enhanced if it can be executed. For this to take place, the notation should have an operational semantics that is equivalent to its declarative semantics, and that can be used to describe the behavior of the interpreter or code generator. It is important that these semantics are formal, because ambiguities in this early specification stage may lead to expensive errors later. The only way to eliminate ambiguity is to be formal.

We should add that this does not imply that the reader of the specification must know all the formal details of the semantics. Meyer [1985] has shown that formal descriptions can be translated in clear and unambiguous natural language descriptions that can be used to communicate the meaning of a specification to people not schooled in formal specifications. Similarly, a diagrammatic specification that has a formal semantics can be used to clearly and unambiguously communicate a meaning to people not schooled in formal specifications. Diagram techniques with formal semantics combine the best of the worlds of semiformal and formal specification [Harel 1992].

Having argued for the need for formality underlying semiformal notation techniques, it is only a small step to argue for simplicity as well. It is all too easy to add numerous annotations, adornments, and extensions to graphical techniques. However, every such addition makes it more difficult to define a formal declarative or operational semantics of the notation. And until a formal semantics has been defined, the extension cannot be used for the job it is intended to perform, that is, communicate a meaning clearly and unambiguously from one person to another. From a research point of view, the way to go is to define the simplest technique possible that still is of use to the specifier, and give this a formal declarative and possible operational semantics. One advantage of simplicity is that it should be relatively easy to define such a semantics. Another advantage, not to be discounted, is that it should be easy for specifiers to remember the meaning and use of the notation—and for different specifiers to remember the same thing. Specifiers in real development projects have enough to think about and should not be bothered with overly complex specification techniques. Not unlike incremental development of software systems, the simple core notation could be incrementally extended to suit the needs of the specifier, all the time taking care that the extensions have a formal semantics. There is no reason to expect that these incremental extensions will be universally useful. More likely, we can expect different extensions to emerge for different domains.

There are a few efforts to define a formal semantics for diagram notations. Most notable is the formal semantics of statecharts used in Statemate [i-Logix 1991b; Harel and Naamad 1996] and of a more recent object-oriented version of statecharts [Harel and Gery 1996, 1997]. Another influential diagram technique with a formal semantics is SDL. Less widely used but very interesting is the formal semantics given to OMT models by Cook and Daniels [1994]. In addition, there are a number of formalizations of structured techniques [France 1992; France and Larrondo-Petrie 1994] and of object-ori-

ented techniques [Bates et al. 1996; Bourdeau and Cheng 1995; Wieringa et al. 1993; Wieringa and Saake 1996]. An important unsolved point in the formalization of object-oriented class models is the precise meaning of behavior specialization. Preliminary investigations are presented by Ebert and Engels [1997], McGregor and Dyer [1993], Schrefl and Stumptner [1995], and Saake et al. [1994]. Møller-Pedersen et al. [1987] describe the solution used in SDL92 and Cook and Daniels [1994] define the solution used in Syntropy. However, it is safe to say that a satisfactory solution has not yet been found. We should stress that formalization of a diagram technique cannot be a useful goal in itself. The formalization must always be combined with a motivation that the diagram technique is useful for some purpose.

This brings us to another interesting research topic, the definition of criteria to choose among the techniques available to the software engineer. It is clear that specification of data-intensive software can benefit from the use of ERDs or class diagrams, and that the specification of control-intensive software can benefit from the use of STDs and communication diagrams. To give more detailed advice, however, the semantics of the notations should be defined precisely. Just as formalization should be guided by the potential uses of the notation, the guidelines for using the notation should be motivated by the formal semantics.

APPENDIX A. A CATALOGUE OF STRUCTURED SPECIFICATION METHODS

In this appendix and the next, we use the bold font to indicate columns in Tables A.I and A.II and italics to indicate entries in these columns.

A.1 SADT

Structured Analysis and Design Technique (SADT) is a method to functionally decompose the activities to be performed by a system into subactivities [Marca and Gowan 1988; Ross 1977, 1985]. SADT uses one diagram technique, the activity diagram (Section 3.3.3), for the representation of external functions, conceptual components (activities), and communication between those components. The highest-level activity diagram represents the overall functionality of the system, its interfaces, and the mechanisms that it uses to realize its functionality. The lower-level diagrams describe the decomposition into subactivities, their communication by means of flows, and the mechanisms used by the subactivities. The subactivities are recursively decomposed until a clear understanding of the activities to be performed by the system is reached. We consider the activities to be conceptual components, so that the activity diagram shows a conceptual decomposition of the system into functional activities.

The original proposal for SADT envisaged a duality of data and activities. Diagrams with the same syntax as activity diagrams were used to represent data and the way they are connected by processes [Ross 1977]. This kind of diagram does not seem to be used any more, and it is not mentioned by Marca and Gowan [1988].

A.2 The Yourdon Systems Method

The Yourdon approach to structured analysis started in the late 1970s with DeMarco [1978], Weinberg [1978], Yourdon and Constantine [1979], and Gane and Sarson [1979]. Flavin [1981] introduced data modeling concerns. Major advances were achieved by McMenamin and Palmer [1984], who introduced the heuristics of essential systems modeling and event partitioning, and Ward and Mellor [1985], who introduced real-time specification techniques. Yourdon [1989] summarized the method. The Yourdon Systems Method (YSM) is a more recent and extensive statement of structured analysis for real-time systems [Yourdon Inc. 1993]. Other ver-

Table A.I. Structured Software Specification Methods

	External communications	External behavior	External functions	Conceptual components	Component functions	Component behavior	Component communication
SADT				• Activity diagram	• Activity diagram		• Activity diagram
YSM	• Context diagram		• System purpose specification • Event-response list	• DFD • ERD	• Minispecs • Decision tables	• Mealy STDs or –tables	• DFD • Function-entity table
Statemate	• Activity charts			• Activity charts	• Imperative specifications	• Statecharts	• Activity charts • Statecharts
SDL	• Message sequence charts			• Block diagrams	• Internal to state diagrams	• SDL state diagrams	• Block diagrams
Information Engineering		• Process dependency diagram	• Function refinement tree	• Entity-relationship diagram • Entity-process matrix			• Process dependency diagram
JSD 83				• SND	• PSD documentation	• PSDs	• SNDs

sions of structured analysis for real-time systems are presented by Gomaa [1993], who gives a number of important design heuristics, and by Hatley and Pirbhai [1987], who add system-level considerations. Shumate and Keller [1992] integrate the Hatley–Pirbhai approach with other approaches to structured analysis.

A YSM model of the **external functions** consists of an informal but concise *specification of the system purpose,* and an *event-response list* (Section 3.4.2). In terms of the function decomposition table, the top row of the table lists all the event-response pairs, each of which is a system transaction. **External communications** are represented by a *context diagram* (Section 3.3.2). If there are too many external entities or interfaces to show on one diagram, then one can show a set of *partial* context diagrams, one for each major subfunction of the system.

The **conceptual decomposition** is represented by a *dataflow diagram* (Section 3.3.1), which represents the activity performed during a system function (an event-response pair). The leftmost column of the function decomposition table thus lists all data stores, data processes, and control processes. In addition, an *entity-relationship diagram* (Section 3.2.1) shows the conceptual structure of the data manipulated during this activity. Dataflow and entity-relationship diagrams would be accompanied by textual specifications noting the functions of the data processes, the contents of the dataflows and data stores, and the conceptual structure of each entity type and relationship. There is also a data dictionary that defines the

Table A.II. Object-Oriented Software Specification Methods

	External communication	External behavior	External functions	Decomposition into objects	Object operations	Object behavior	Object communication
Shlaer & Mellor 88, 92	• Object communication model			• Information model	• Executable code • Action data flow diagrams	• State machines	• Object communication model • Object access model
Coad & Yourdon 90				• Class diagram	• Service specifications		• Class diagram
Booch 91, 94				• Class diagram	• Part of class specification	• State transition diagram	• Object diagram • Timing diagram
OMT 91	• Event trace diagrams			• Object model	• Functional model	• Dynamic model	• Event trace diagrams
OOSE/Objectory 92, 95	• Use case model		• Use case model	• Analysis object model	• Analysis object specifications	• State machines	• Interaction graphs
Martin & Odell 92, 95				• Object diagram			• Event diagrams
Embley *et al.* 1992				• Object-relationship model	• Object specifications	• Object-behavior model	• Object interaction model
Champeaux 93				• Static object model.	• Action specifications	• Behavior diagrams	• Object interaction diagrams
Firesmith 93	• Context diagram		• Statement of purpose	• Semantic net • Classification diagram • Composition diagram	• Textual specifications	• State transition diagrams	• Object interaction diagrams • Control flow diagrams • Timing diagrams
Fusion 94		• System life cycle model	• System operation model	• System object model	• Object operation specifications		• Object interaction graphs
SOMA 94				• Class diagram	• Operation specifications	• State model	• Class diagram

Continued on next page

meaning of all names that occur in the DFD and ERD.

Different authors state different rules for connecting an ERD with a DFD in a coherent YSM model. The source most explicit about this, Yourdon Inc. [1993, pp. 309–314], states merely that the ERD must describe the conceptual structure of all data manipulated by the system. One may set up the model in such a way that for each entity type and relationship there is a data store that

Table A.II. Continued

Continued from previous page

	External communication	External behavior	External functions	Decomposition into objects	Object operations	Object behavior	Object communication
ROOM 94				• Actor class specifications	• Protocol class specifications	• RoomCharts	• Actor class diagram • Message sequence chart
MOSES 94	• Scenario descriptions		• User requirements specification	• Object/Class model • Inheritance model	• Service specifications	• Object charts	• Event model
Syntropy 94				• Object type model	• Formal specification	• Object behavior model	
OMT 95	• Event trace diagrams • Use cases	• Event trace diagrams	• Use cases	• Class diagram • Object diagrams	• Declarative specifications	• Dynamic model	• Event trace diagrams • Object interaction diagram • Object-oriented DFDs
Mainstream Objects 95	• Transaction sequence diagram		• Transaction sequence diagram	• Object structure diagram		• Object life cycle diagram • Activity flow diagram	• Object interaction diagram: Fence • Object interaction diagram: Net
BON 95		• Scenario charts	• System purpose chart • Event charts	• System purpose charts • Cluster charts • Class charts • Structured static diagrams			• Event charts • Object creation charts • Dynamic diagrams
Fusion 96	• Use case diagrams • System interaction diagram • Scenario diagrams	• System interaction diagram • Scenario diagrams	• Use case diagrams • System operation model	• System object model	• Object operation specifications		• Object interaction graphs

Continued on next page

contains the instances of that type. This is the way in which Figures 7 and 5 are related. However, there may be a many–many relationship between the components of an ERD and the data stores. In addition, an ERD may describe the conceptual structure of the data that survive a single system transaction (stored in a persistent data store) or that exist only during a transaction (passing through a dataflow or stored in a temporary data store).

Table A.II. Continued

Continued from previous page							
	External communication	External behavior	External functions	Decomposition into objects	Object operations	Object behavior	Object communication
Octopus 96	• System use case diagram • System context diagram	• Subsystem event specifications • Subsystem state models • Subsystem event significance tables	• System use case diagram • Subsystem operation specifications	• Subsystem object model			
OOram 96	• External scenario view • External collaboration view	• External scenario view • External collaboration view	• Area of concern view • Stimulus-response view	• Internal collaboration view	• Interface view • Method specification view		• Internal collaboration view • Internal scenario view
UML 97	• Use case model • Collaboration diagram • Sequence diagram	• Collaboration diagram • Sequence diagram	• Use case model	• Class diagram	• Object operation specifications	• State diagrams • Activity diagrams	• Collaboration diagrams • Sequence diagrams

Only continuing data and control processes have **behavior** over time. Instantaneous processes do not have an internal state, so that they always give the same response to the same input. The behavior of a continuing data process is specified by a minispec, which can follow any format, such as a pre- or postcondition specification for the initial action when the process is started, and a pre- or postcondition specification for the continuing transformation [Yourdon Inc. 1993, p. 250]. The interface of the transformation must correspond to the interface of the data process in the DFD. A continuing control process is specified by a Mealy machine (STD or state transition table). The events and actions of the Mealy machine must correspond to the interface of the control process in the DFD.

YSM also allows the specification of the behavior of entities. For each entity type in the ERD, an entity state diagram can be defined, which is a Mealy machine that regulates the accesses made to the instances of the entity type.

The **functions** of the conceptual components are specified differently for different kinds of components. There are altogether five kinds of components of DFDs: data stores, instantaneous and continuing data processes, and instantaneous and continuing control processes. The functions offered by data stores are the ability to create, read, update, and delete records in the store. There is no need for a specification of these data store functions. Instantaneous data processes compute the same function every time they are executed. This function is specified in a *minispec,* which may be declarative or imperative (Section 3.4.3). This technique can also be used for specifying continuing data processes. The elementary functions offered by continuing control processes are the transitions in their state machine and these are specified in an STD or a state transition table. An instantaneous control process is simply a *decision table* and can be specified as such. In all these cases, the interface of the minispec or decision table must match the interface of the corresponding process in the DFD. There is as yet no

definitive precise formulation of this coherence rule.

The **communication** between components is represented by the DFD. We can draw one DFD fragment for each event-response pair, which corresponds to one column in the function decomposition table. This is what is done in event partitioning. Each such DFD shows the communications that take place between the conceptual components during the transaction.

The accesses that are made to a data store during a system transaction are summarized in YSM by a *function-entity table,* which lists external functions against entity types and relationships, and in the entries shows whether instances of these types are created, read, updated, or deleted during the transaction. This is a restriction of the function decomposition table to the data store components.

A.3 Statemate

Statemate is a software toolkit and a notation for structured analysis. A Statemate specification consists of *activity charts* (Section 3.3.4), *statecharts* (Section 3.5.7), and *module charts.* Compared to the Yourdon school of structured analysis, activity charts play the role of DFDs, statecharts play the role of Mealy machines, and module charts play the role of architecture diagrams of Hatley and Pirbhai [1987] and Shumate and Keller [1992] (not treated in this survey). In the interest of executability of the specification, actions performed by statecharts are defined *imperatively.* The execution semantics of Statemate models is defined formally [i-Logix 1991b; Harel and Naamad 1996].

The language of statecharts is considerably richer than the Mealy machine notation used in other methods of structured analysis. Statemate does not contain a notation corresponding to ERDs. The way in which Statemate puts activity charts and statecharts together into a coherent model differs in details from the way this is done in other structured analysis methods but on the whole, the model structure is the same as the way DFDs and Mealy machines are put together in YSM. The general remarks made for YSM, minus the discussion of ERDs, therefore also apply to Statemate.

A.4 SDL and Message Sequence Charts

The development of SDL (Specification and Description Language) started in 1972 and resulted in 1976 in a standard issued by the CCITT (International Telegraph and Telephone Consultative Committee). Since then, revisions and extensions were defined first by the CCITT and then by the ITU (International Telecommunication Union) at four-year intervals [Belina et al. 1991; Bræk and Haugen 1993]. Sarraco and Tilanus [1987] and Belina and Hogrefe [1988–1989] give tutorials on SDL 88 and Færgemand and Olsen [1994] on SDL 92. A rationale for the object-oriented features of SDL 92 is given by Møller-Pedersen et al. [1987]. SDL is a language, not a method. However, some method guidelines are published by the ITU as an appendix to the SDL definition [ITU 1993]. Saracco et al. [1989] and Turner [1993] give many examples of and heuristics for the use of SDL.

The **external behavior** and **external communication** of a system can be described by *message sequence charts* (MSC) [ITU 1994]. These are sequence diagrams (Section 3.3.8), used to describe sequences of possible interactions between the system and its external entities. Each such sequence is an illustration of the behavior in which the system must be able to engage.

The **conceptual decomposition** of a system is represented by a *block diagram* (Section 3.3.7). In the same way as hierarchical DFDs, block diagrams represent the system as a hierarchical decomposition into blocks. Each leaf block in the block decomposition tree contains one or more processes. As pointed out in Section 3.3.7, block diagrams may be used to represent the system decomposition or they may be an

organizing principle of the specification of external system behavior.

At each level, the **component communication** is represented by channels between blocks or signal routes between processes. (SDL allows implicit signal routes that are not represented visually.) Communication through channels may involve delay. Communication between processes goes through signal routes, which experience no delay. However, each process has an unbounded input queue of messages that are waiting to be processed. Upon sending, a signal immediately arrives at the input queue of its receiver but may then have to wait for some time before it is processed. As mentioned in Section 3.5.8, processes have an alternative means of communication, defined by means of exported variables.

The **behavior** of a process is defined by means of an *SDL state diagram* (Section 3.5.8). The interface of a state diagram must match the interface of the corresponding process in the block diagram. The behavior of each leaf block is defined totally by the joint behavior of its processes and the way these are connected by signal routes to each other and to the block environment. The behavior of each nonleaf block follows from the behavior of its subblocks and the way they are connected by channels to each other and to the block environment.

The semantics of SDL models is defined in terms of an underlying virtual machine upon which the model can be executed.

A.5 Information Engineering

Information engineering (IE) is a structured method for modeling the information needs of a business, specifying information systems that meet these needs, and designing, constructing and introducing these systems in the business [Martin and Finkelstein 1981; Martin 1989a,b,c]. In this survey, we deal with information system specification only. To deliver the information systems that a business needs, IE first abstracts from current and future systems and identifies business functions and their information needs independently from the way these are realized. This is represented by a *function refinement tree* (Section 3.4.1), which refines the business mission into business functions and refines these into business processes. The subject domain of the business—the part of the world that the business must maintain data about—is represented by an entity-relationship diagram. The function tree and ER diagram are then correlated by means of an entity-function table that shows the use that each business process makes of business data. The table can be partitioned into coherent parts called business areas. Each business area is modeled in more detail by a *process dependency diagram* (Section 3.5.9) and a more detailed *ER diagram* (Section 3.2.1). These more detailed processes and entities are correlated by an entity-process table. If the business area is too big to be implemented by one information system, it is partitioned into several information systems called business systems.

Thus, the **external functionality** of an information system is represented by a function refinement tree and its **conceptual decomposition** is specified by an ER diagram and a process dependency diagram. The process dependency diagram also specifies the **external behavior** of the information system. It does not specify the behavior of the system components but shows how pieces of externally observable processes are ordered in time. These processes are components of the conceptual decomposition of the system. Finally, the process dependency diagram also shows how these processes **communicate.**

An interesting phenomenon of IE is that the conceptual decomposition of the information system corresponds to the decomposition of the subject domain of the business. The conceptual decomposition of the information supply function of the business is found by decomposing

the subject domain. The ERD of the subject domain is then used to partition the information supply function into business areas and these into business systems. This data-orientation is a central tenet of IE: whereas business procedures change rapidly, data types are relatively stable [Martin 1989a, p. 58]. This principle is also central to JSD, discussed next, and of object-oriented methods.

A.6 Jackson System Development (JSD)

Jackson System Development (JSD) is an elaboration of Jackson Structured Programming (JSP) to the system level [Jackson 1975, 1983]. In JSD, the subject domain of a software system is modeled as a set of communicating entities. Each subject domain entity is modeled as a finite state machine by means of a *process structure diagram* (Section 3.5.2). Entities may have attributes, which can be updated only by an action in the PSD of the entity. Entities communicate synchronously by sharing common actions. Entities are always classified into types and the PSD is always specified for a type.

In order to define a **decomposition** of the system, for each UoD entity, the software product contains a surrogate that performs the same process as the UoD entity. ("Surrogate" and "PSD" are not JSD terms.) Each surrogate process is described initially by the same PSD as its corresponding subject domain entity. When system functions are specified, these PSDs may be extended. In addition, the software product contains function processes, also described by PSDs. These communicate with the surrogate processes through various kinds of connections, represented by a *system network diagram* (SND) (Section 3.3.6).

There are thus two kinds of conceptual components, surrogate processes and function processes. It is argued by Jackson [1983, pp. 11–12] that this distinction leads to more maintainable systems than models based upon functional decomposition. For example, changes to functions will often be constrained to the function processes and leave the surrogates unaffected.

The **component functions** are the atomic operations by which a surrogate participates in an external system function; they are different from the atomic operations by which a function process participates in an external function. During the execution of an external system function F, a surrogate s may perform one action, which is a leaf node of its PSD. This action is an atomic operation that can stand on its own as an update transaction. During an external function, on the other hand, a function process may completely execute its PSD. For example, if the function process is a query that should report on the state of a number of surrogates, then the PSD for that function will specify imperatively how the function will search through the surrogates. Whenever the function process does something, this function executes its entire PSD. In other cases, a function process may be long-running, so that each time it does not execute its entire PSD but only a part of it. In general, the actions in a function process cannot stand on their own as system transactions but are only part of an imperative specification of a system transaction.

The **behavior** of a surrogate is specified by a PSD that describes its lifecycle. Some function processes may also be long-running, in which case their PSD also describes the behavior of the function process over time; that is, it orders several atomic pieces of behavior of the function process in time.

Communication between components is represented by an SND, one for each external system function. This SND corresponds to a column in the function decomposition table. The structure of the SND is very similar to the structure of an event-partitioned DFD that could be drawn for the external function.

APPENDIX B. A CATALOGUE OF OBJECT-ORIENTED SPECIFICATION METHODS

B.1 Shlaer–Mellor 1988, 1992

Shlaer and Mellor [1988] define an object-oriented variant of information modeling [Chen 1976; Flavin 1981]. Shlaer and Mellor [1992] extend this with techniques for the specification of object behavior and object communication, and add the concepts of domain and subsystem. Additional information on the Shlaer–Mellor method is given by Lang [1993], Shlaer and Mellor [1989], and Shlaer and Lang [1996].

Shlaer and Mellor partition complex systems into domains, where a domain is defined as a part of the world with its own conceptual space of rules and policies. Example domains are the implementation domain (programming languages, operating system, database management systems, etc.), the service domain (containing utility functions for user interfaces, data archiving, etc.), and the application domain (e.g., a juice plant). The application domain is what we call the subject domain in this survey. Large domains can be further partitioned into subsystems that have close cohesion and loose coupling. Each subsystem is modeled as a collection of communicating objects.

The **decomposition into objects** of a subsystem is represented by an *information model,* which uses a variant of the ERD notation (Section 3.2.1). There are special objects called timers that can be created to count the time remaining to a deadline. Timers are not specified in the information model.

Object behavior is represented by a *state model,* which is a Moore state machine (Section 3.5.6). Thus, a transition between states is triggered by an event arriving at the object and upon arrival in a state, an action may be performed.

Object operations are actions associated with a state. Each action is specified by means of a piece of *executable code* associated with the state (Section 3.4.3). An action may read or update the state of any object in the subsystem, it may create an object, and it may send an event to any object (including itself) or to an entity outside the subsystem [Shlaer and Mellor 1992, p. 45]. Actions take time to execute in OOA. Each state model corresponds to a row in the function decomposition table, for it shows the behavior of a component.

In addition to the code associated with a state, an *action dataflow model* (ADFD) is specified for each action. This is a dataflow model that is used to specify the data processing done by an action. A data store in an ADFD corresponds to an object class (it holds all instances of the class), to a timer object, or to the current time. A process in the ADFD corresponds to an elementary computation performed during the action. If an ADFD showed what one object action does, it would correspond to a cell in the function decomposition table. However, the ADFD also shows how other objects are accessed (read or write), so it represents a part of what goes on in the life of other objects as well. In fact, ADFDs are used to specify object actions in the same way as YSM uses a DFD to specify external system functions. ADFDs thus introduce functional decomposition in the model, where the decomposition criterion is very fine-grained: each action of each object is functionally decomposed into an ADFD. This makes the model conceptually difficult, because each ADFD separates data storage and data processing in a way that goes counter to the object-oriented philosophy. The reason for introducing this difficulty is not clear, because actions are already specified by means of code associated to a state.

Object communication is shown by means of two models. The *object communication model* (OCM) shows the messages sent and received by the state machines of the objects. This is represented by an object communication diagram (Section 3.3.5). Communication between objects via events is asynchronous. Events received from one object

are processed in the order in which they are sent, but events received from different objects may be processed in any order [Shlaer and Mellor 1992, p. 107]. External entities are included in the OCM, so that it also shows the **external communications** of the system. Except for the representation of external entities, all information in the OCM can be derived from the information in the state model. There is one OCM for the entire specification.

The second kind of communication model is the *object access model* (OAM), which shows accesses to object data stores made by processes in the ADFDs for the actions of an object. This is represented by a directed graph in which the nodes represent object classes and the edges represent synchronous access of one object by another. These communications are synchronous. The OAM can be derived from the ADFDs. There is one OAM for the entire model.

B.2 Coad–Yourdon 1990

Coad and Yourdon [1990] use a kind of *class diagram* (Section 3.2.2) to represent the **conceptual decomposition** of the system into objects. The same diagram is also used to specify **object communications.** Large class diagrams can be partitioned into chunks called subjects. A class icon contains the name of the class and the attributes and operations (called services) of the objects of the class. *Service specifications* are declarative natural language descriptions of **object operations.**

B.3 Booch 1991, 1994

Booch represents the structure of a software system by means of a class diagram and the behavior of the objects by means of state diagrams. Booch [1991] uses simple state diagrams, whereas Booch [1994] uses a statechart-like notation. Communication is represented by Booch [1991] by timing diagrams and by Booch [1994] by sequence diagrams. Timing diagrams resemble sequence diagrams turned on their sides. We discuss the 1994 version of the Booch notation.

The **conceptual decomposition** of the software into objects is represented by a *class diagram* (Section 3.2.2). Booch uses a dotted cloud symbol to represent the classes. As usual, this can be annotated with the class name, attributes, and operations, possibly adorned with symbols that indicate how the class is to be implemented in a programming language. Classes can be parameterized and Booch distinguishes metaclasses. The diagram can be annotated by various kinds of constraints. The full notation contains a considerable number of icons and symbols, the meaning of which is informally explained.

Object behavior is specified by a *state transition diagram,* for which Booch uses a statechart-like notation (Section 3.5.7). Booch does not use Cartesian products to represent parallelism because he claims that parallelism is already sufficiently represented by the parallelism of objects. His notation is therefore not able to represent intra-object parallelism.

Classes can be specified textually in a programming language. **Object operation specifications** are part of this specification.

Object communication is represented in two ways. An *object diagram* is a collaboration diagram (Section 3.3.9) that can be used to show the flow of messages during a scenario of the system's behavior. They can be annotated with dataflows, the communication mechanisms to be used, visibility relations between the objects, and constraints on the links between the objects in the diagram. Communication can be synchronous or asynchronous. Several other kinds of communication are recognized. The other way to represent communication is by means of an *interaction diagram,* which is a sequence diagram (Section 3.3.8). These contain roughly the same information as object diagrams.

B.4 OMT 1991, 1995

The object modeling technique (OMT) was introduced by Loomis et al. [1987] and popularized by Rumbaugh et al. [1991]. We refer to this version of OMT as OMT91. A significant update was published several years later [Rumbaugh 1995c,a,b). We refer to this as OMT95.

The **decomposition** of the system into objects is represented by the *object model,* which is a class diagram (Section 3.2.2). The **behavior** of instances of classes is represented by the *dynamic model* which is a statechart variant (Section 3.5.7). Events in the state model can be represented as operations in the object model [Rumbaugh et al. 1991, p. 110]. Statecharts are inherited by subclasses but the precise mechanism by which this happens is not defined [Rumbaugh et al. 1991, p. 111]. We already observed in Section 5 that behavior inheritance is one of the unsolved problems of object-oriented specification.

Object operations are defined by the *functional model.* This is a DFD (Section 3.3.1) that shows which computations are performed by the system during an external system function. Just like ADFDs in the Shlaer–Mellor method (Appendix B.1) the DFD introduces a functional decomposition into the model. However, where Shlaer–Mellor advise us to define a functional decomposition for each action, OMT gives no heuristic for the functional decomposition. The connection between the functional model and the other two models is ill-defined. Objects in the object model are viewed as components of the system, and object operations correspond to the processes in the DFD [Rumbaugh et al. 1991, p. 137], however, objects in the object model are viewed as *external entities* in the functional model and object operations correspond to data flows to and from external entities [Rumbaugh et al. 1991, p. 138]. This matter has never been cleared up and in OMT95, the functional model as we know it from OMT91 disappeared.

OMT95 can be viewed as a halfway station between OMT91 and the UML. The *object model* of OMT95 contains some extensions with respect to OMT91 [Rumbaugh 1995b]. It now contains a class diagram as well as object diagrams, with which instances can be represented. It also contains conventions to represent parameterized classes and to represent subsystems.

In the *dynamic model,* scenarios play a more prominent role and use cases have been adopted. Event trace diagrams can be annotated with constraints on the time between two events. In the statecharts that describe object behavior, states now can have local variables. Interobject concurrency is described more explicitly as follows. Like an SDL process, each atomic object has a queue of input events, but unlike SDL processes, OMT allows aggregate objects that have an input queue for each component object [Rumbaugh 1995a, p. 10]. Two other extensions to the state model are the inclusion of entry and exit actions in states and of timing marks along transitions.

The *functional model* has undergone a major revision in OMT95 [Rumbaugh 1995c]. For most system functions (called operations in OMT), a simple declarative pre- or postcondition specification in terms of the system objects will do. For complicated functions, OMT95 offers object interaction diagrams and object-oriented dataflow diagrams (OODFDs). An object interaction diagram is a kind of collaboration diagram and can be used to show the sequence of messages that implement a function. It corresponds to a column in the function decomposition table. An OODFD is a directed higraph that represents objects as nodes. These contain nodes that represent object functions (cells in our function decomposition table) and nodes that represent object attributes. Object attribute and function nodes are connected by directed edges that show which attributes are read or

written by an operation and what the dataflows between the operations are. These flows may cross object boundaries. This certainly improves the definition of the relationship between the functional model and the object model. Nevertheless, the meaning of OODFDs is still defined rather informally. In the UML, they have disappeared from the scene.

B.5 Objectory 1992, 1995

Objectory is the commercialized version of object-oriented software engineering (OOSE) [Jacobson et al. 1992]. It is described in the documentation that comes with the Objectory Case tool [Objectory AB 1995b,c].

External functions of the systems are represented by *use cases* (Section 3.4.4).

To prepare for a **conceptual decomposition** of the system, a *domain object model* is made, which represents subject domain entities and their relationships. The domain object model also represents communications between domain entities [Objectory AB 1995b, p. 82]. The 1995 version of Objectory uses a nonstandard notation for this, but it is to be expected that later versions of the UML will use the UML class diagram notation. The conceptual decomposition of the system is represented by the *analysis object model.* This distinguishes three kinds of objects. Entity objects contain information about some phenomenon in the subject domain, interface objects handle communication with the environment, and control objects encapsulate use-case-specific behavior. There is likely to be a close correspondence between objects in the domain object model and entity objects in the analysis model. Control objects are similar to JSD's function processes.

Atomic **object operations** are specified informally by means of *object specifications.*

The **behavior** of objects can be specified by an SDL-like state transition diagram (Section 3.5.8), called the *object behavior model.*

Communication between objects can be illustrated by means of sequence diagrams (Section 3.3.8), called the *object interaction model.*

B.6 Martin–Odell 1992, 1995

The first version of the Martin–Odell method is described in a single book [Martin and Odell 1992]. The second version is described in three books, of which Martin and Odell [1995] give an overview of the notation used. An important difference between the 1992 and the 1995 description is that in the latter, the description of the model structure is separated from a discussion of the different possible notations that could be used to express this structure. The bulk of the Martin and Odell [1995] uses object and event diagrams to represent object structure and behavior, but in addition a variety of structure and behavior diagraming techniques is discussed.

The **conceptual decomposition** of a system is represented by an *object model,* which is a class diagram-like technique (Section 3.2.2).

Behavior is analyzed as a dynamic structure of events that cause objects to change state. Events are brought about by operations and may trigger other operations, provided that the guard for these other operations is satisfied. This guard is called a control condition for the triggered operation. The relationships between events, triggers, and operations is represented in an *event diagram,* which is in the same league as process dependency diagrams (Section 3.5.9). Event diagrams do not show the behavior of a single object but "meaningful" behavior structures that are not necessarily localized into a single object [Martin and Odell 1995, p. 118]. As a result, each event diagram represents an aspect of external system behavior as well as some of the communication between objects. An event diagram could, for example, be used to represent

the behavior of objects during the execution of a system function (column of the function decomposition table) or of an entire use case.

Martin and Odell [1995] discuss the use of various dialects of class diagrams to represent object structure, different kinds of finite state diagrams to represent object behavior, and various kinds of sequence and collaboration diagrams to represent object communication.

B.7 Embley 1992

Embley et al. [1992] represent the **conceptual decomposition** of a system by the *object-relationship model,* which is a kind of class diagram (Section 3.2.2).

Object behavior is represented by the *object-behavior model,* which uses a diagram convention called state nets, that combine some features of Petri nets (not treated here) and Mealy state diagrams (Section 3.5.5). A state net can also be used to represent intra-object concurrency. A transition can require an object to be in several states simultaneously and it can lead to a set of simultaneous states. A transition can also require that an object must be in at least one of a set of states and it can lead to one out of a set of possible states.

Object communication is represented by *interaction diagrams,* which are hypergraphs that can show synchronous and asynchronous message-passing between objects. Communication can be binary (one sender and one receiver) or broadcast, and unidirectional or bidirectional.

Object behavior and interaction diagrams can be annotated by timing constraints. In the behavior diagram, one can constrain the time that an object can reside in a state, the time it takes for a trigger to take effect, the time it takes to execute an action, the time that an entire object transition can take, and the time it takes to execute a scenario through the state machine. In the interaction diagram, one can constrain the time it takes for a message to arrive at its destination.

All diagrams can be abstracted to higher-level overview diagrams. Thus, one can represent high-level object classes, relationship sets, state sets, transition sets, and interaction sets.

B.8 De Champeaux 1993

Champeaux et al. [1993] represent **system decomposition** by their own brand of class diagrams called a *static object model.* A special feature of their object model is that they recognize ensembles, which are a kind of aggregate objects that encapsulate their components. Ensembles are similar to ROOM actors that also have their own behavior defined in addition to the behavior defined for their multiple components (Appendix B.11). **Object behavior** is specified by a Mealy-like notation extended with guards and object interaction. **Object operations** can be described in informal text, formal pre- and postconditions, or even DFDs [Champeaux et al. 1993, p. 69]. **Object communication** is represented by directed graphs called *interaction diagrams.* There are notations for synchronized bidirectional communication and asynchronous unidirectional communication.

B.9 Firesmith 1993

Firesmith [1993] defines an elaborate set of notations for specifying systems, subsystems, and objects. Systems and subsystems are called assemblies and subassemblies, respectively. Where the Coad–Yourdon notation (Appendix B.2) is by far the simplest of the reviewed notations, the Firesmith notation is the most complex of the reviewed notations. For each assembly, the **external functions** are described globally by a *statement of purpose* and the **external communications** are specified by means of a *context diagram.* In addition, a decomposition into subassemblies is defined and for each subassembly, the purpose is specified. Table A.II lists the tech-

niques for specifying subassemblies, plus the statement of purpose and context diagram for the entire assembly.

The **decomposition** of a subassembly into objects is represented by a directed graph called a *general semantic net,* which shows individual objects and their relationships. This is supplemented by a *decomposition diagram* that shows aggregation hierarchies of objects. Object classes are shown in a separate *classification diagram.* All of this can be supplemented by textual specifications in a programming-language-like format.

Object behavior is specified by *state transition diagrams.* **Object communication** is specified by *interaction diagrams,* which are directed graphs whose nodes represent individual objects. The flow of data and control through objects can be specified by a DFD-like notation called a *control flow diagram.* A sequence of interactions can be illustrated by a *timing diagram,* which resembles a sequence diagram (Section 3.3.8). The diagram conventions are all different from the notations reviewed in this survey.

B.10 SOMA 1994

The Semantic Object Modeling Approach (SOMA) defined by Graham [1994] is an extension of the Coad–Yourdon notation (Appendix B.2). Other acknowledged influences are Booch [1994], Champeaux et al. [1993], Desfray [1992], Jacobson et al. [1992], Henderson-Sellers and Edwards [1994], and Rumbaugh et al. [1991].

The **decomposition** of the system is specified by an extension of the Coad–Yourdon *class diagrams.* One extension is that in addition to a class name, attributes, and operations, a class box may contain business rules applicable to instances of that class. Another extension is that SOMA allows fuzzy objects, which may have fuzzy attribute values, fuzzy rule sets, and only partial inheritance from superclasses. As in Coad–Yourdon, class diagrams are also used to specify **communication** between objects (called usage in SOMA).

Object operations are specified using any possible technique, ranging from state diagrams, JSD's process structure diagrams, or DFDs [Graham 1994, p. 249]. No information is given on how these techniques should be combined with class diagrams. Graham does give the advice to use these techniques on an object-by-object basis and not for global system description.

Object behavior is specified using *state diagrams.* SOMA allows Mealy diagrams, statecharts, and PSDs as a notation.

B.11 ROOM 1994

The real-time object-oriented modeling (ROOM) method partitions the system into a hierarchical collection of components called actors [Selic et al. 1994]. An actor is a logical component of a system that can be active concurrently to the other actors in the system. If the actor is a software object, it has its own thread of control. If it is a physical object, it can behave independently of other objects. In both cases, it must have a clearly defined purpose [Selic et al. 1994, pp. 66, 150]. If an actor A is decomposed into subactors, interaction of another actor with these subactors is only possible through A.

A ROOM model represents the **decomposition** of the system as a decomposition hierarchy of actors. This is represented by means of *actor class diagrams,* which are undirected higraphs in which the nodes represent actors and the edges represent communications. The communication links between actors are called bindings. Actor class diagrams are very much like SDL block diagrams (Section 3.3.7), except that actor class diagrams allow the specification of behavior for blocks at any level in the hierarchy. Second, actor class diagrams require the explicit specification of the communication mechanisms for each binding [Selic et al. 1994, p. 159], something which is defined for

block diagrams by the SDL standard. Like SDL models, ROOM models are executable.

Like SDL, ROOM allows the partitioning of large systems into layers. The difference between layers and decomposition into subactors is that the actors in a lower layer continue to exist when actors in a higher layer are destroyed. By contrast, when a compound actor is destroyed, its subactors are destroyed as well [Selic et al. 1994, p. 195].

Actor **behavior** is represented by a statechart-like technique called *RoomCharts.* Like SDL processes, each actor has an unbounded queue of messages that have been received but not yet processed [Selic et al. 1994, pp. 219, 292].

Communication between actors is represented by the actor class diagrams, mentioned previously, and can be illustrated by *message sequence charts* (Section 3.3.8). Communication can be synchronous or asynchronous [Selic et al. 1994, pp. 292–293]. In asynchronous communication, the sender continues processing immediately after sending a message. In synchronous communication, the sender waits until it has received a reply from the receiver. This reply is treated with higher priority than other received messages; that is, it bypasses any messages waiting to be processed. After receiving the reply, the sender continues its own processing.

B.12 Fusion 1994, 1996

Fusion is a method defined after an analysis of early object-oriented methods like OMT, Booch, and Objectory [Bear et al. 1990; Coleman et al. 1992, 1994; Hayes and Coleman 1991]. These methods were analyzed using a dialect of the formal specification language VDM as a tool. Significant updates to Fusion were published in 1996 [Coleman 1996; Malan et al. 1996].

Fusion defines a **decomposition** of a system by first specifying a *domain model,* which is an ER-like diagram of the subject domain. The system decomposition is then found by outline of the system boundary in the domain model. Everything inside the boundary is part of the system; everything outside the boundary is part of the environment.

External functions are called *system operations* in Fusion, and are specified declaratively by means of pre- and postcondition specifications (Section 3.4.3), a technique borrowed from VDM specification theory. The 1996 update extends this with *use case diagrams* (Section 3.4.4) to describe external functionality.

External system behavior is represented by means of regular expressions over system operations. In the 1996 update these were replaced by *system interaction diagrams,* which are collaboration diagrams (Section 3.3.9) that show sequences of interactions between the system and the environment. Alternatively, *sequence diagrams* (Section 3.3.8) can be used, which are called scenario diagrams in Fusion. Together with use case diagrams, these diagrams also show aspects of **external communication**).

Object behavior is not specified in Fusion at the conceptual level. Instead, during design, external operations are decomposed into object operations by means of collaboration diagrams, called *object interaction graphs.* These also show the **communication** between objects. **Object operations** are specified declaratively by means of pre- and postconditions.

B.13 MOSES 1994

"MOSES" stands for methodology for object-oriented software engineering of systems (Henderson-Sellers and Edwards 1994] and defines a product lifecycle from initial conception to product maturity, in which each change of the product is performed according to a process lifecycle. MOSES also defines notations to be used for the deliverables of the process as well as activities that yield these deliverables. The process is structured according to the fountain

model [Henderson-Sellers and Edwards 1990] and consists of planning, investigation, specification, and review phases. The investigation phases yields a *user requirements specification* and system *scenarios,* both written in natural language. Both of these are a way of specifying **external system functions.**

The **conceptual decomposition** of a software system is specified using an *object/class model,* which is a class diagram-like technique that can be used to specify a decomposition into individual objects or into classes of objects. The inheritance structure can be specified separately by an *inheritance model.* MOSES provides some extra notations for the object/class model to represent programming language constructs such as client-server and friend relationships, and exceptions.

Object operations are described by service specifications, which can take two forms. *Contracts* specify a service offered by an object by means of pre- and postconditions (Section 3.4.3). Alternatively, a service can be specified imperatively by means of a *service structure model,* which is a kind of structure chart adapted to object-oriented programming, proposed by Page-Jones et al. [1990]. Either way, a service specification specifies an individual object action, which corresponds to a cell in the function decomposition table.

Object behavior is represented by *objectcharts,* which is a statechart-like notation (Section 3.5.7). **Object communication** is specified by an *event model,* which is a kind of collaboration diagram (Section 3.3.9). Communication is synchronous in MOSES [Henderson-Sellers 1992, p. 56]. There is a guideline that an event model should correspond to an external system function [Henderson-Sellers 1992, p. 300]. An event model thus corresponds to a column in the function decomposition table.

Complex system specifications can be simplified by partitioning the specification into sheets and/or the system into subsystems. Complexity can be further reduced by partitioning classes or subsystems into layers [Henderson-Sellers 1992, p. 250].

B.14 Syntropy 1994

Syntropy [Cook and Daniels 1994] is a combination of OMT and statecharts, extended with some other notations from Booch, all given a precise semantics by means of the formal specification language Z [Spivey 1989]. In the process of providing this semantics, many details of these notations are clarified. For example, Syntropy is the only reviewed method that gives a semantics to behavior inheritance [Cook and Daniels 1994, pp. 206 ff.]. In addition, the notation is supplemented with modeling guidelines and a particular view of the development process.

Syntropy distinguishes the essential model, the specification model, and the implementation model of a software system. The *essential model* is a model of the subject domain. This is modeled as a collection of objects and events. Objects in the subject domain have properties, which can only change as the result of an event in the subject domain. Events may change the properties of more than one object. Events are instantaneous and simultaneously observable everywhere. The essential model consists of a *type view,* which represent the types of objects in the subject domain, and a *state view,* which represents the way objects change as a result of events. The type view is represented by an OMT-like *object diagram,* which is a kind of class diagram (Section 3.2.2), supplemented with Z specifications for types and invariants. In the state view, a *state model* is defined for each object type that provides the details of object creation, the list of events to which objects of this type can respond, pre- and postconditions of these events, and a statechart (Section 3.5.7) that shows how objects of this type respond to events. Each event must appear in at least one statechart but may appear in more. The pre- and postconditions for each event are specified in the formal

specification language Z. Preconditions are the conditions that must exist for the event to occur. In the essential model, it is impossible for an event to occur when the preconditions are not satisfied.

The term "essential model" in Syntropy should not be confused with the same term as used in Structured Analysis. In Structured Analysis, it is used to refer to an implementation-independent software specification, which corresponds to the software specification model of Syntropy.

The *specification model* of Syntropy describes the states that the software can be in and the way that it responds to stimuli by changing state and producing responses. Each stimulus is an event that is broadcast to all objects in the model simultaneously. As in the essential model, all events are instantaneous. This means that all the consequences of an external stimulus are completed before any further external stimuli occur. If an external stimulus triggers several events in several statecharts, these all occur in an unspecified order before the next stimulus is recognized. In contrast to the essential model, if a precondition is not satisfied, the event may occur but its effect is undefined. The specification model is described by the same techniques as the essential model, that is, a type view and a state view. The entries for Syntropy in Table A.II describe the specification model. The type view of the specification model represents the **conceptual decomposition** of the software into objects and the state view represents **object behavior.**

The *implementation model* describes the flow of control of the software. Stimuli are mapped to messages and all message-passing and method executions take time. The implementation model must deal explicitly with concurrency, persistence, finite resources, errors, exceptions, and the like. The implementation model uses collaboration diagrams called mechanisms to specify the sequence of messages that implements external system functions.

To build a specification model, the system boundary must be defined by defining external entities, called agents in Syntropy. These may be modeled as part of the essential model. For each event in the essential model it must be decided whether it is to be detected by the software, generated by the software, or ignored. In addition, the specification model shows how undesirable events are handled. These were simply impossible in the essential model.

To manage complexity, Syntropy partitions the software into subsystems called domains. A *concept domain* is a model of the subject domain. A software system may have more than one concept domain. An *interaction domain* provides the mechanism for keeping the concept domain and the world in step and an *infrastructure domain* provides general-purpose abstraction that can be used by objects in the other domains. Agents with which the system must communicate may be represented as software objects in the specification model, for example, in the interaction domain or in a concept domain. The type view of a concept domain will be very similar to the type view of the essential model [Cook and Daniels 1994, p. 262]. The state views of the essential and specification models usually differ. The reason is that essential model statecharts show event sequencing, whereas specification model statecharts show how responses are generated by software.

B.15 Mainstream Objects 1995

Mainstream Objects is a combination of techniques from different object-oriented methods [Yourdon et al. 1995]. **External functions** are modeled by use cases (Section 3.4.4), called *transaction sequences* in this approach. The **conceptual decomposition** of a system is represented by an *object structure diagram,* which is a kind of class diagram (Section 3.2.2). The overall decomposition of the system into sub-

systems is represented by a *system overview diagram.* This shows the subsystems and the services they offer to each other. Object structure, behavior, and interaction are then shown for each subsystem.

Object behavior is represented by *object life cycle diagrams,* which is a statechart-like formalism (Section 3.5.7). **Object communication** is specified using sequence diagrams or collaboration diagrams, called *fence* and *net interaction diagrams,* respectively (Sections 3.3.8 and 3.3.9). Finally, the flow of activities in the system can be represented by an *activity diagram,* which is a kind of process dependency diagram (Section 3.5.9). These show behavior over time, not restricted to any object.

B.16 Business Object Notation 95

In the business object notation (BON), a number of textual structured natural language specifications, called charts, are combined with two kinds of diagrams [Waldén and Nerson 1995]. **External behavior** is described by *scenario charts,* which are step-by-step descriptions of scenarios. **External functions** are described by a *system purpose chart.* In addition to a description of the system purpose, somewhat similar to YSM's system purpose specification, these specify a decomposition of the system into clusters of classes. Each cluster is briefly described. In addition, *system event charts* list external events to which the system must respond and, for each event, the classes that play a role in recognizing the event and producing the response. These are similar to event-response specifications of YSM but system event charts give considerably less information.

The **system decomposition** is specified in two parts. The decomposition into clusters is shown on the system purpose chart and the decomposition of a cluster into classes is shown in cluster charts. Relationships between classes can be represented in a *static structure diagram.* This includes inheritance relations between classes. Static structure diagrams are directed graphs but use a nonstandard convention.

Communication between components is represented by *dynamic diagrams,* which resemble collaboration diagrams (Section 3.3.9). In addition, each event chart lists the classes that play a role in recognizing and responding to the event, and there are object creation charts that specify which classes of objects can create which classes of objects.

B.17 Octopus 1996

Octopus is an extension of Fusion with notions useful for the specification and design of embedded real-time systems [Awad et al. 1996]. Like Fusion 96, Octopus specifies **external system functions** by means of *use cases,* specified declaratively by means of pre- and postconditions. The use cases are not represented by use case diagrams but by OMT-like object diagrams, that can show aggregation relationships between use cases. **External communication** is specified by a *context diagram,* again using an OMT-like object diagram. This allows one to show relationships between external entities, such as aggregation and communication relationships.

Octopus borrows the domain concept of the Shlaer–Mellor method to define subsystems. Example domains are the application domain, the device domain, and the implementation domain. Octopus recognizes a special kind of subsystem called a hardware wrapper that hides the hardware from the application subsystem. The partitioning into subsystems is represented by an OMT-like object diagram, showing in particular the aggregation relationships between subsystems. The Octopus entry in Table A.II lists the techniques to specify external communication and functions, and the techniques to specify subsystems.

For each subsystem the **decomposition into objects** is represented by an OMT-like object diagram. This resembles a class diagram (Section 3.2.2). The

subsystem responds to events by performing operations. The **operations** of the subsystem are described by declarative specifications called *operation sheets.* In addition to pre- and postconditions, these also mention the objects involved in responding to the event. Each event received by a subsystem is broadcast to all objects in the system and all ensuing processing is considered to be instantaneous.

Subsystem behavior is described by a *subsystem state model,* which is a statechart (Section 3.5.7). Following the Fusion philosophy, statecharts are not defined for object types. Instead, a statechart is specified for every interesting aspect of subsystem behavior. Statecharts define subsystem states and show the effect of events on this state. The external behavior entry for Octopus in Table A.II lists techniques to specify external subsystem behavior, which is why statecharts are mentioned in this entry.

For each event to which the subsystem must respond, an *event significance* table is made, which shows for each possible state of the subsystem how critical the event is. The more critical an event is, the more urgent it is that the subsystem respond immediately when the event occurs.

As in Fusion, the mechanism by which objects implement a subsystem operation is represented in the design phase by *object interaction graphs,* which are collaboration diagrams (Section 3.3.9).

Octopus has an interesting similarity with YSM, used as a method to specify subsystems. External subsystem behavior is specified in both by means of a context diagram, pre- and postcondition specifications of events to which the subsystem must respond, and state transition models of subsystem behavior (control processes in YSM). Even the ERD in YSM plays a similar role to the object diagram in Octopus. Both represent the conceptual decomposition of the subsystem. The difference lies in the interpretation of this as data stores in YSM and as objects in Octopus. Furthermore, Octopus models do not contain data processes and they do not focus on dataflows, as YSM models do.

B.18 OOram 1996

The central concept in object-oriented role analysis and modeling (OOram) is the concept of a role, defined as a position that a set of objects can occupy in a pattern of collaborating objects [Reenskaug et al. 1996, p. 22]. A role has an identity and has behavior and state that is encapsulated. It is an archetypical representation of the objects occupying the corresponding position in the object system [Reenskaug et al. 1996, p. 7]. The collaboration pattern characteristic for a role is represented by a kind of collaboration diagram. This is called a *collaboration view* and can be used to represent the collaboration of the system with its environment as well as the collaboration of objects in the system to realize an external system function. Several collaboration patterns can be specified for a system, one from each relevant point of view. These different patterns can then be synthesized into a overall collaboration model of all objects in the system performing all their different functions. OOram offers a number of synthesis operations for this.

OOram recognizes a number of system views, not all of which will be needed in all projects. **External behavior** and **communication** are represented by a sequence diagram (Section 3.3.8) called the *external scenario view.* This can be supplemented by the *external collaboration view,* which uses collaboration diagrams (Section 3.3.9). The same kind of diagram can also be used to represent the communication between objects in the system. **External functions** are represented by a global description of the system purpose in natural language, called the *area of concern view.* System functions are described briefly in the *stimulus-response view,* which consists of a list of stimuli and corresponding responses, annotated

with brief comments. This resembles a simplified form of an event-response list (Section 3.4.2).

System decomposition into objects as well as **communication** between objects are represented by the *internal collaboration view,* which is again a collaboration diagram. As mentioned previously, communication between components can also be represented by sequence diagrams. **Object functions** are specified in a programming-language-like way called the *interface view*. This lists all the interfaces of all objects. In addition, a *method specification view* can be made, which extends the sequence diagrams of the internal scenario view with annotations of the methods executed by an object as a result of the reception of a message.

B.19 UML 1997

Version 1.0 of the Unified Modeling Language (UML) arose as a result of a joint effort of Booch, Jacobson, and Rumbaugh to unify the existing notations for object-oriented software specification. It combines the notations used in the Booch method, Objectory, and OMT, applies some simplifications, and extends this with new features and diagrams. However, the basic structure of the notations used in these methods remains recognizable. The intention is that it will be used as a diagram convention in those methods and probably in other methods as well. In January 1997, the UML version 1.0 was submitted to the Object Management Group (OMG) along with several other proposals. The submitters then negotiated a joint submission, which led to Version 1.1 of the UML [Rational 1997b,c]. This was accepted in November 1997 by the OMG as standard notation for object-oriented specifications.[3] At the time of this writing, Booch, Jacobson, and Rumbaugh all announced books on UML 1.1 and several companies announced that they will adopt the UML as standard notation in their method.

All of the diagrams used in the UML have been explained in Section 3. Here, we briefly review the terminology of the UML. **External behavior** and **communication** of a system is represented by *collaboration* and *sequence diagrams* (Sections 3.3.8 and 3.3.9), in which the system figures as one of the communicating entities. **External functions** are represented by *use case diagrams* and their attendant documentation. These also illustrate a part of external system communication. The **conceptual decomposition** is represented by a *static structure diagram,* which is a class diagram (Section 3.2.2) extended with the possibility of representing individual objects as well as classes. **Component behavior** is represented by statecharts (Section 3.5.7) adapted to object-oriented use, and **communication** between components is represented by *collaboration* and *sequence diagrams* (Sections 3.3.8 and 3.3.9). The UML defines a special kind of statechart called an *activity diagram* to represent dependency between activities. This is similar to a process dependency diagram (Section 3.5.9). Activity diagrams can be used to specify procedure-like object behavior, object operations, or use cases.

The total UML notation is quite complex. It contains by far the largest number of icons and symbols for adorning, commenting, and otherwise labeling the diagrams. This makes the definition of a formal semantics a daunting task. It also makes the notation hard to use. Methods that use this notation should give heuristics for the good use of all these features that are consistent with the semantics of the notations—once these semantics are defined. The problem is compounded by the fact that the UML contains constructs that allow different methods to extend the notation with new features. Even if we assume each method gives a precise semantics to its UML extensions, this may cause a proliferation of UML dialects, understood only by their inventors.

[3] See http://www.omg.org for details.

Despite these misgivings, the survey of this article has shown that there is considerable convergence in the notations used by the different methods. Agreeing on a standard notation with a well-defined semantics would allow practitioners to concentrate on solving the customer's problem instead of on learning yet another notation. The future will show whether the UML will develop into the standard that fits this need or whether it will turn out to be a generic template whose meaning is in the eye of the beholder and that can be customized beyond recognition.

ACKNOWLEDGMENTS

The survey benefited from detailed comments by Michael Jackson and Frank Dehne. The anonymous referees gave useful comments for further improvement.

REFERENCES

AWAD, M., KUUSELA, J., AND ZIEGLER, J. 1996. *Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion.* Prentice-Hall, Englewood Cliffs, NJ.

BAETEN, J. AND WEIJLAND, W. 1990. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18, Cambridge, University Press, New York.

BATES, B., BRUEL, J.-M., FRANCE, R., AND LARRONDE-PETRIE, M. 1996. Guidelines for the formalizing Fusion object-oriented analysis models. In *Advanced Information Systems Engineering (CAiSE'96),* P. Constantopoulos, J. Mylopoulos, Y. Vassiliou, Eds., LNCS 1080, Springer, 222–233.

BEAR, S., ALLEN, P., COLEMAN, D., AND HAYES, F. 1990. Graphical specification of object-oriented systems. In *Object-Oriented Programming: Systems, Languages and Applications/ European Conference on Object-Oriented Programming.* Ottawa, ACM, N. Meyrowitz, Ed., SIGPLAN *Not. 25,* 10, 28–37.

BEECK, M. V. D. 1994. A comparison of Statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems,* H. Langmaack, W. d. Roever, and J. Vytopil, Eds., LNCS 863, Springer, 128–148.

BELINA, F. AND HOGREFE, D. 1988–1989. The CCITT-specification and description language SDL. *Comput. Netw. ISDN Syst. 16,* 311–341.

BELINA, F., HOGREFE, D., AND SARMA, A. 1991. *SDL with Applications from Protocol Specification.* Prentice-Hall, Englewood Cliffs, NJ.

BLANCHARD, B. AND FABRYCKY, W. 1990. *Systems Engineering and Analysis.* Prentice-Hall, Englewood Cliffs, NJ.

BOOCH, G. 1991. *Object-Oriented Design with Applications.* Benjamin/Cummings, Redwood City, CA.

BOOCH, G. 1994. *Object-Oriented Design with Applications, Second Edition.* Benjamin/Cummings, Redwood City, CA.

BORGIDA, A., MYLOPOULOS, J., AND REITER, R. 1995. On the frame problem in procedure specifications. *IEEE Trans. Softw. Eng. 21,* 785–798.

BOURDEAU, R. AND CHENG, B. 1995. A formal semantics for object model diagrams. *IEEE Trans. Softw. Eng. 21,* 10, 799–821.

BRÆK, R. AND HAUGEN, Ø. 1993. *Engineering Real-Time Systems.* Prentice-Hall, Englewood Cliffs, NJ.

CHAMPEAUX, D. D. AND FAURE, P. 1992. A comparative study of object-oriented analysis methods. *J. Object-Oriented Programs,* 21–33.

CHAMPEAUX, D. D., LEA, D., AND FAURE, P. 1993. *Object-Oriented System Development,* Addison-Wesley, Reading, MA.

CHEN, P.-S. 1976. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst. 1,* 9–36.

COAD, P. AND YOURDON, E. 1990. *Object-Oriented Analysis.* Yourdon Press/Prentice-Hall, Englewood Cliffs, NJ.

COLEMAN, D. 1996. Fusion with use cases: Extending Fusion for requirements modeling. Available at URL: http:/www.hpl.hp.com/fusion/index.html.

COLEMAN, D., ARNOLD, P., BODOFF, S., DOLLIN, C., GILCHRIST, H., HAYES, F., AND JEREMAES, P. 1994. *Object-Oriented Development: The FUSION Method.* Prentice-Hall, Englewood Cliffs, NJ.

COLEMAN, D., HAYES, F., AND BEAR, S. 1992. Introducing Objectcharts or how to use Statecharts in object-oriented design. *IEEE Trans. Softw. Eng. 18,* 1, 9–18.

COOK, S. AND DANIELS, J. 1994. *Designing Object Systems: Object-Oriented Modelling with Syntropy,* Prentice-Hall, Englewood Cliffs, NJ.

DAHL, O.-J. AND NYGAARD, K. 1966. SIMULA—an ALGOL-based simulation language. *Commun. ACM 17,* 7, 403–412.

DAVIS A. 1993. *Software Requirements: Objects, Functions, States.* Prentice-Hall, Englewood Cliffs, NJ.

DEMARCO, T. 1978. *Structured Analysis and System Specification.* Yourdon Press/Prentice-Hall, Englewood Cliffs, NJ.

DESFRAY, P. 1992. *Ingénerie des Objects Approche Classe-Relation Applications à C++.* Editions Massons, Paris.

EBERT, J. AND ENGELS, G. 1997. Specialization of object life cycle definitions. Submitted.

ECKERT, G. 1996. Improving the analysis stage of the Fusion method. In *Object-Oriented Development at Work: Fusion in the Real World,* R. Malan et al. Eds., Prentice-Hall, Englewood Cliffs, NJ, 276–313.

EMBLEY, D., JACKSON, R., AND WOODFIELD, S. 1995. OO systems analysis: Is it or isn't it. *IEEE Softw. 12,* 3, 19–33.

EMBLEY, D., KURTZ, B., AND WOODFIELD, S. 1992. *Object-Oriented Systems Analysis: A Model-Driven Approach.* Prentice-Hall, Englewood Cliffs, NJ.

FÆRGEMAND, O. AND OLSEN, A. 1994. Introduction to SDL 92. *Comput. Netw. ISDN Syst. 26,* 1143–1167.

FICHMAN, R. AND KEMERER, C. 1992. Object-oriented and conventional analysis and design methodologies: Comparison and critique. *Computer 25,* 22–39.

FIRESMITH, D. 1993. *Object-Oriented Requirements Analysis and Logical Design: A Software Engineering Approach.* Wiley, New York.

FLAVIN, M. 1981. *Fundamental Concepts of Information Modeling.* Yourdon Press, Englewood Cliffs, NJ.

FOWLER, M. 1994. Describing and comparing object-oriented analysis and design methods. In Object Development Methods, A. Carmichael, Ed., SIGS Books, Denville, NJ, 79–109.

FRANCE, R. 1992. Semantically extended data flow diagrams: A formal specification tool. *IEEE Trans. Softw. Eng. 18,* 4, 329–346.

FRANCE, R. AND LARRONDO-PETRIE, M. 1994. From structured analysis to formal specification: State of the theory. In *Proceedings of the 1994 ACM Computer Science Conference.*

GANE, C. AND SARSON, T. 1979. *Structured Systems Analysis: Tools and Techniques.* Prentice-Hall, Englewood Cliffs, NJ.

GOLDSMITH, S. 1993. *Real-Time Systems Development.* Prentice-Hall, Englewood Cliffs, NJ.

GOMAA, H. 1993. *Software Design Methods for Concurrent and Real-Time Systems.* Addison-Wesley, Reading, MA.

GRAHAM, I. 1994. *Migrating to Object Technology.* Addison-Wesley, Reading, MA.

HALL, P., OWLETT, J., AND TODD, S. 1976. Relations and entities. In *Modelling in Database Management Systems,* G. Nijssen, Ed., North-Holland, 201–220.

HAREL, D. 1979. *First Order Dynamic Logic.* LNCS 68, Springer.

HAREL, D. 1988. On visual formalisms. *Commun. ACM 31,* 514–530.

HAREL, D. 1992. Biting the silver bullet. *Computer 25,* 1, 8–20.

HAREL, D. AND GERY, E. 1996. Executable object modeling with statecharts. In *Proceedings of the Eighteenth International Conference on Software Engineering,* 246–257.

HAREL, D. AND GERY, E. 1997. Executable object modeling with statecharts. *Computer 30,* 7, 31–42.

HAREL, D. AND KAHANA, C.-A. 1992. On statecharts with overlapping. *ACM Trans. Softw. Eng. Methodol. 1,* 4, 399–421.

HAREL, D. AND NAAMAD, S. 1996. The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol. 5,* 293–333.

HAREL, D. AND PNUELI, A. 1985. On the development of reactive systems. In *Logics and Models of Concurrent Systems,* K. Apt, Ed., NATO ASI Series, Springer, 477–498.

HAREL, D., PNUELI, A., SCHMIDT, J., AND SHERMAN, R. 1987. On the formal semantics of statecharts. In *Proceedings, Symposium on Logic in Computer Science,* Computer Science Press, New York, 54–64.

HATLEY, D. AND PIRBHAI, I. 1987. *Strategies for Real-Time System Specification.* Dorset House, New York.

HAYES, F. AND COLEMAN, D. 1991. Coherent models for object-oriented analysis. In *Object-Oriented Programming: Systems, Languages and Applications/European Conference on Object-Oriented Programming,* A. Paepcke, Ed., SIGPLAN *Not. 25,* 11, 171–183.

HENDERSON-SELLERS, B. 1992. *A Book of Object-Oriented Knowledge.* Prentice-Hall, Englewood Cliffs, NJ.

HENDERSON-SELLERS, B. AND EDWARDS, J. 1990. The object-oriented systems life cycle. *Commun. ACM 33,* 9, 143–159.

HENDERSON-SELLERS, B. AND EDWARDS, J. 1994. *Book Two of Object-Oriented Knowledge.* Prentice-Hall, Englewood Cliffs, NJ.

HODGSON, R. 1994. Contemplating the universe of methods. In *Object Development Methods,* A. Carmichael, Ed., SIGS Books, Denville, NJ, 111–132.

HOPCROFT, J. AND ULLMAN, J. 1979. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA.

HUTT, A. 1994a. *Object Analysis and Design: Comparison of Methods.* OMG/Wiley, New York.

HUTT, A. 1994b. *Object Analysis and Design: Description of Methods.* OMG/Wiley, New York.

I-LOGIX 1991a. The Languages of STATEMATE. Tech. Rep., i-Logix Inc., Burlington, MA. To be published as D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The STATEMATE Approach.*

I-LOGIX 1991b. The Semantics of statecharts. Tech. Report, i-Logix Inc., Burlington, MA.

IVARI, J. 1995. Object-orientation as structural, functional and behavioural modelling: A comparison of six methods for object-oriented analysis. *Inf. Softw. Technol. 37,* 3, 155–163.

ITU 1992. *Specification and Description Language (SDL).* International Telecommunications Union (formerly CCITT), Revised recommendation Z.100.

ITU 1993. Z.100 Appendix I: SDL Methodology Guidelines. Tech. Rep., International Telecommunication Union.

ITU 1994. *Criteria for the Use and Applicability of Formal Description Techniques: Message Sequence Charts (MSC).* International Telecommunication Union. Z.120 (03/93).

JACKSON, M. 1975. *Principles of Program Design.* Academic Press.

JACKSON, M. 1983. *System Development.* Prentice-Hall, Englewood Cliffs, NJ.

JACKSON, M. 1995. *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices.* Addison-Wesley, Reading, MA.

JACOBSON, I., CHRISTERSON, M., JOHNSSON, P., AND OVERGAARD, G. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Prentice-Hall, Englewood Cliffs, NJ.

LANG, N. 1993. Shlaer–Mellor object-oriented analysis rules. *Softw. Eng. Not. 18,* 1, 54–58.

LOOMIS, M., SHAH, A., AND RUMBAUGH, J. 1987. An object modeling technique for conceptual design. In *European Conference on Object-Oriented Programming,* J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, Eds. LNCS 276, Springer, Paris, 192–202.

MALAN, R., LETSINGER, R., AND COLEMAN, D., EDS. 1996. *Object-Oriented Development at Work: Fusion in the Real World.* Prentice-Hall, Englewood Cliffs, NJ.

MARCA, D. AND GOWAN, C. 1988. *SADT: Structured Analysis and Design Technique.* McGraw-Hill, New York.

MARTIN, J. 1989a. *Information Engineering, Book I: Introduction.* Prentice-Hall, Englewood Cliffs, NJ.

MARTIN, J. 1989b. *Information Engineering, Book II: Planning and Analysis.* Prentice-Hall, Englewood Cliffs, NJ.

MARTIN, J. 1989c. *Information Engineering, Book III: Design and Construction.* Prentice-Hall, Englewood Cliffs, NJ.

MARTIN, J. AND FINKELSTEIN, C. 1981. *Information Engineering.* Savant Institute, Carnforth, England.

MARTIN, J. AND MCCLURE, C. 1985. *Diagramming Techniques for Analysts and Programmers.* Prentice-Hall, Englewood Cliffs, NJ.

MARTIN, J. AND ODELL, J. 1992. *Object-Oriented Analysis and Design.* Prentice-Hall, Englewood Cliffs, NJ.

MARTIN, J. AND ODELL, J. 1995. *Object-Oriented Methods: A Foundation.* Prentice-Hall, Englewood Cliffs, NJ.

MCGREGOR, J. AND DYER, D. 1993. Inheritance and state machines. *Softw. Eng. Not. 18,* 4, 61–69.

MCMENAMIN, S. AND PALMER, J. 1984. *Essential Systems Analysis.* Yourdon Press/Prentice-Hall, Englewood Cliffs, NJ.

MEYER, B. 1985. On formalism in specifications. *IEEE Softw.* 6–26.

MILNER, R. 1980. *A Calculus of Communicating Systems.* LNCS 92, Springer.

MØLLER-PEDERSEN, B., BELSNES, D., AND DAHLE, H. 1987. Rationale and tutorial on OSDL: An object-oriented extension of SDL. *Comput. Netw. ISDN Syst. 13,* 97–117.

MONARCHI, D. AND PUHR, G. 1992. A research typology for object-oriented analysis and design. *Commun. ACM 35,* 9, 35–47.

OBJECTORY AB 1995a. *Objectory: Introduction, Version 3.6.*

OBJECTORY AB 1995b. *Objectory: Requirements Analysis, Version 3.6.*

OBJECTORY AB 1995c. *Objectory: Robustness Analysis, Version 3.6.*

OMG 1996. Object analysis and design RFP-1. OMG TC Document ad/96-05-01—rev.1, 6/6/96. Available at URL http://www.omg.org/public-doclist.html.

PAGE-JONES, M., CONSTANTINE, L., AND WEISS, S. 1990. Modeling object-oriented systems: The uniform object notation. *Comput. Lang. 7,* 10, 69–87.

PETERSOHN, C., HUIZING, C., PELESKA, J., AND DE ROEVER, W.-P. 1994. Formal semantics for Ward and Mellor's TRANSFORMATION SCHEMAS. In *Sixth Refinement Workshop,* Workshops in Computing, BCS-FACS, D. Till, Ed., Springer Verlag, 14–41.

PNUELI, A. AND SHALEV, M. 1991. What is in a step: On the semantics of statecharts. In *Theoretical Aspects of Computer Software,* T. Ito and A. Meyer, Eds. LNCS 526, Springer, 244–264.

RATIONAL 1997a. *Unified Modeling Language: Notation Guide, Version 1.0.* Rational Software Corporation, Santa Clara, CA. Available at URL http://www.rational.com/ot/uml.html.

RATIONAL 1997b. *Unified Modeling Language: Notation Guide, Version 1.1.* Rational Software Corporation, Santa Clara, CA. Available at URL http://www.rational.com.uml/1.1/.

RATIONAL 1997c. *Unified Modeling Language: Semantics, Version 1.1.* Rational Software Corporation, Santa Clara, CA. Available at URL http://www.rational.com/uml/1.1/.

REENSKAUG, T., WOLD, P., AND LEHNE, O. 1996. *Working with Objects: The OOram Software*

Engineering Methodology. Computer Society Press.

ROSS, D. 1977. Structured analysis (SA): A language for communicating ideas. *IEEE Trans. Softw. Eng. SE-3,* 1, 16–34.

ROSS, D. 1985. Applications and extensions of SADT. *Computer 18,* 4, 25–34.

RUMBAUGH, J. 1995a. OMT: The dynamic model. *J. Object-Oriented Program. 7,* 9, 6–12.

RUMBAUGH, J. 1995b. OMT: The functional model. *J. Object-Oriented Program. 8,* 1, 10–14.

RUMBAUGH, J. 1995c. OMT: The object model. *J. Object-Oriented Program. 7,* 8, 21–27.

RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. 1991. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ.

SAAKE, G., HARTEL, P., JUNGCLAUS, R., WIERINGA, R., AND FEENSTRA, R. 1994. Inheritance conditions for object life cycle, diagrams. In *Formale Grundlagen für den Entwurf von Informationsystemen,* U. Lipeck and G. Vossen, Eds. Institut für Informatik, Universität Hannover, Germany 79–88. Informatik-Berichte Nr. 03/94.

SARACCO, R., SMITH, J., AND REED, R. 1989. *Telecommunications Systems Engineering Using SDL,* North-Holland.

SARRACO, R. AND TILANUS, P. 1987. CCITT SDL: Overview of the language and its applications. *Comput. Netw. ISDN Syst. 13,* 65–74.

SCHREFL, M. AND STUMPTNER, M. 1995. Behavior consistent extension of object life cycles. In *ER'95*.

SELIC, B., GULLEKSON, G., AND WARD, P. 1994. *Real-Time Object-Oriented Modeling*. Wiley, New York.

SHLAER, S. AND LANG, N. 1996. The Shlaer-Mellor method: The OOA96 report, version 1.0. Tech. Rep., Project Technology, Inc., Berkeley, CA 984710. Available at URL http://www-.projtech.com/cgi/pdf/ooa96.pdf.

SHLAER, S. AND MELLOR, S. 1988. *Object-Oriented Systems Analysis: Modeling the World in Data*. Prentice-Hall, Englewood Cliffs, NJ.

SHLAER, S. AND MELLOR, S. 1989. An object-oriented approach to domain analysis. *ACM SIGSOFT Softw. Eng. Not. 14,* 5, 66–77.

SHLAER, S. AND MELLOR, S. 1992. *Object Lifecycles: Modeling the World in States*. Prentice-Hall, Englewood Cliffs, NJ.

SHUMATE, K. AND KELLER, M. 1992. *Software Specification and Design: a Disciplined Approach for Real-Time Systems*. Wiley, New York.

SPIVEY, J. 1989. *The Z Notation: A Reference Manual*. Prentice-Hall, Englewood Cliffs, NJ.

SUTCLIFFE, A. 1991. Object-oriented systems development: Survey of structured methods. *Inf. Softw. Technol. 33,* 6, 433–442.

TURNER, K., ED. 1993. *Using Formal Description Techniques: An Introduction to ESTELLE, LOTOS and SDL*. Wiley, New York.

WALDÉN, K. AND NERSON, J.-M. 1995. *Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems*. Prentice-Hall, Englewood Cliffs, NJ.

WARD, P. AND MELLOR S. 1985. *Structured Development for Real-Time Systems*. Prentice-Hall/Yourdon Press, Englewood Cliffs, NJ.

WEGNER, P. 1992. Dimensions of object-oriented modeling. *Computer 25,* 10, 12–20.

WEINBERG, V. 1978. Structured Analysis. Yourdon Press, Englewood Cliffs, NJ.

WIERINGA, R. 1996. *Requirements Engineering: Frameworks for Understanding*. Wiley, New York.

WIERINGA, R. AND SAAKE, G. 1996. A formal analysis of the Shlaer–Mellor method: Towards a toolkit for formal and informal requirements specification techniques. *Req. Eng. 1,* 106–131.

WIERINGA, R., JUNGCLAUS, R., HARTEL, P., SAAKE, G., AND HARTMANN, T. 1993. OMTROLL—object modeling in Troll. *Proceedings of the International Workshop on Information Systems—Correctness and Reusability (ISCORE'93),* Udo W. Lipeck and G. Koschorrek, Eds. Institut für Informatik, Universität Hannover, Germany, 267–283.

WIRFS-BROCK, R., WILKERSON, B., AND WIENER, L. 1990. *Designing Object-Oriented Software*. Prentice-Hall, Englewood Cliffs, NJ.

YOURDON, E. 1989. *Modern Structured Analysis*. Prentice-Hall, Englewood Cliffs, NJ.

YOURDON, E. AND CONSTANTINE, L. 1979. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Englewood Cliffs, NJ.

YOURDON, E., WHITEHEAD, K., THOMANN, J., OPPEL, K., AND NEVERMANN, P. 1995. *Mainstream Objects: An Analysis and Design Approach for Business*. Prentice-Hall, Englewood, Cliffs, NJ.

YOURDON, INC. 1993. *Yourdon™ Systems Method: Model-Driven Systems Development*. Prentice-Hall, Englewood Cliffs, NJ.

Received May 1997; Accepted January 1998