Defining Specialization for Process Models

George M. Wyner (gwyner@bu.edu)
Boston University School of Management
(ph) 617-353-4153

Jintae Lee (jintae@colorado.edu)
College of Business. Univ. of Colorado
(ph) 303-492-4149

Index Terms: Organizational Design, Process Design, Process Innovation, Business Process Reengineering, Process Models, Process Representation, Specialization, Inheritance, Data Flow Diagram, State Diagram, Object-Oriented Paradigm, Electronic Commerce, Electronic Markets

Abstract

Object-oriented analysis and design methods take full advantage of the object specialization hierarchy when it comes to modeling the objects in a system. When modeling system behavior, however, system analysts continue to rely on traditional tools such as state diagrams and dataflow diagrams. While such diagrams capture important aspects of the processes they model, they offer limited guidance as to the ways in which a process can be improved. In this paper we extend the notion of specialization to process representations and identify a set of transformations which, when applied to a process description, always result in specialization. We analyze specific examples in detail and demonstrate that such a use of specialization is not only theoretically possible, but shows promise as a method for categorizing and analyzing processes. This paper makes two contributions toward answering this question: first, it articulates a formal definition of

process specialization which is compatible with object specialization but allows us to reason specifically in terms of process representations. Second, it develops the concept of the "specializing transformation" as a means for systematically generating and exploring process alternatives. We illustrate these results by applying them to two commonly used representations: the state diagram and the dataflow diagram. We identify a number of apparent inconsistencies between process specialization and the object specialization which is part of the object-oriented approach. We demonstrate that these apparent inconsistencies are superficial and that the approach we take is compatible with the traditional notion of specialization.

Biography of Authors

George Wyner received an A.B. in Mathematics from Harvard College in 1981 and a Ph.D. in Management from the Sloan School of Management at MIT in 2000. He is currently Assistant Professor of Information Systems at the Boston University School of Management. Wyner's research centers on the modeling, classification, and analysis of organizational processes in pursuit of systematic technology-enabled organizational innovation. He is a long time member of the Process Handbook Project at MIT's Center for Coordination Science, which is using ideas from coordination theory and object-oriented programming to build an on-line handbook of business processes constructed so as to facilitate the invention of new organizational forms.

Jintae Lee received the B.A. in Mathematics from the Univ. of Chicago, Chicago, IL. in 1979, M.A. in Psychology from Harvard University, Cambridge, and Ph.D. in Electrical Engineering and Computer Science from MIT in 1991. His research interest is in process representation and categorization. He is currently leading the effort in the Process Interchange Format project, whose aim is to define an interlingua for sharing process descriptions. Currently the project includes active members from major process research groups at several universities (MIT, Stanford, Univ. of Hawaii, Univ. of Toronto, Univ. of Edinburgh) and companies. He is also an active member of the MIT Process Handbook project. His previous research includes decision rationale management systems and their use in requirement engineering. Dr. Lee is a member of IEEE, AAAI, and ACM.

Defining Specialization for Process Models

1. INTRODUCTION

As the literature on object-oriented analysis and design attests, specialization of objects is a powerful source of advantage for the design as well as the implementation of information systems (Booch 1991, Coad and Yourdon 1990, De Champeaux 1991, De Champeaux et al. 1990, Maksay and Pigneur 1991, Rogers 1991, Rumbaugh et al. 1991, Taivalsaari 1996, Takagaki and Wand 1991) For example, the specialization hierarchy, in which each object inherits the features of its parent and modifies them incrementally, promotes comprehensibility, maintainability, and reusability.

When modeling system behavior, however, system analysts continue to rely on traditional tools such as state diagrams and dataflow diagrams. While such diagrams capture important aspects of the processes they model, they offer limited guidance as to the ways in which a process can be improved.

Malone et al. (1999) have argued that this limitation of the current approach to information systems design can be addressed by employing a specialization hierarchy of *processes*. In addition to the benefits of comprehensibility and reusability, a process specialization hierarchy offers two major advantages from the organizational design perspective. First, it supports systematic generation of design alternatives. Variants of an existing process can be generated systematically and then evaluated along the dimensions of specialization. This *generativity*[1] is

[1] There is recent precedent for employing the term "generativity" to refer to the capacity of a system to generate novel expressions, combinations, ideas, or productions. See for example (Alexander 1979; Malone et al. 1999).

especially important in the absence of tools that support the design generation stage (Alexander 1979). Second, it offers an organizational framework in which to index and search for relevant processes (Malone et al. 1999). This support for locating relevant processes is especially important in the context of a large database of reusable process models, such as the Process Handbook, designed to support component-based process modeling and design.

The potential value of such an approach is especially promising in light of dramatic changes introduced into organizations by the rise of new information technologies, most recently web computing and the resulting development of electronic commerce. In this paper we focus on the specialization of *processes*. We argue that a clear understanding of when one process is a special case of another and a mastery of the means by which specialized processes can be generated systematically will offer significant support for a more effective exploration of the new design territory opened up by these technologies, since it is arguably processes which must be transformed if business is to tap the potential of these technologies. Once generated by specialization, process variants can be explored and tested for their fit to the new environment (Malone et al 1999).

Implementing such a process hierarchy, however, will require a clear definition of what it means for one process to be a specialization of another, and some guidelines as to how to go about specializing a process in practice.

One obvious approach to this problem is to treat the process as a class with a set of attributes and then to specialize processes in the same way that objects are specialized: by subtyping attributes and adding new attributes. It is not obvious, however, how this approach applies to process descriptions, which are typically complex aggregates of nodes and arcs. Consider, for example, the two state diagrams in Figure 1. Diagram A contains two states R and S. Diagram B is identical except that a third state T has been added. Following the usual approach to

specialization, we might argue that the diagram with the additional feature, diagram B, is a specialization of diagram A. However, one might also maintain that diagram A is the specialization, since diagram B includes all the behaviors of diagram A, but not vice versa, and thus diagram A is a special case of diagram B. Any approach to specializing process representations must yield an operationalization that can explain such a puzzle.

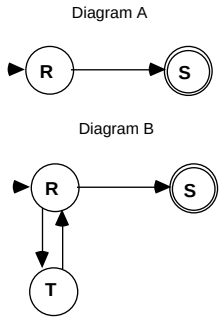

Figure 1. Which diagram is the specialization?

In this paper, we propose a definition of process specialization that resolves this puzzle and supports organizational design by enabling the systematic generation and location of design alternatives. We do not claim at this point that our results generalize beyond the two representations discussed (state diagrams and dataflow diagrams) even though we believe that such general results will be possible. Nor do we claim to be defining a new paradigm in the sense that object-oriented analysis and design is one. In fact, the major goal of this paper is to define *process* specialization so that process representations as well as object representations can be subjected to existing object-oriented methods especially in the context of organizational design.

The rest of the paper proceeds as follows. Section 2 develops a general framework for process specialization. Section 3 applies this approach to state diagrams, deriving a set of transformations which, when applied to any state machine, result in a specialization. Section 4 illustrates the potential benefits of this approach by means of a restaurant information system

example. Section 5 extends the analysis to dataflow diagrams and Section 6 provides an example of how a simple dataflow diagram can be specialized. Section 7 compares this approach to related work. Section 8 identifies and resolves a number of apparent inconsistencies between this approach to process specialization and the approach taken in the object-oriented paradigm. Finally, Section 9 summarizes the results and suggests directions for future research.

2. PROCESS SPECIALIZATION

Consider a system consisting of one or more objects of interest. This may be an existing system which is to be modeled or a new system which is to be designed. Such a system has characteristics which may evolve over time. These changes in the system constitute its *behavior*. We define the *execution set* of a system as the set of all possible behaviors associated with that system. A *process class* describes a set of such systems in terms of their execution sets. A system whose execution set is consistent with a process class is an *instance* of that process class and may be said to *realize* that process. The set of all instances of a process class is referred to as the *extension* of that process class.

Then a process class P` is said to be a *specialization* of a class P if every instance of P` is also an instance of P, but not necessarily vice versa.

2.1. Extension Semantics

There are many methods by which a process class can describe the execution sets of its instances. For example, a class might be defined as including all systems whose execution sets are supersets of some "minimal execution set," —i.e. whose execution sets must include at least all the behaviors specified by the minimal execution set. In contrast, one might define a process class as including all systems whose execution sets are subsets of some "maximal execution set"

– i.e. whose execution sets can include at most the behaviors specified by the maximal execution set. We will refer to this relationship between a process class and its extension as the *extension semantics* of the particular process representation.

For example, in Figure 1, above, the process class represented by diagram B is a specialization of the class represented by diagram A under minimal execution set semantics, because B refers to all systems whose execution sets include at least all the behaviors specified in the diagram and this is clearly a subset of the collection of all systems whose execution sets need include only the behaviors specified in diagram A. In other words, under minimal execution set semantics, each transition represents a constraint, and the more constraints, the *smaller* the extension.

Conversely, under maximal execution set semantics, the process class represented by diagram A is a specialization of that represented by B because A refers to all systems which exhibit a subset of the behaviors in the diagram while B refers to the larger collection of systems which may include any of the additional executions described in diagram B. In other words, in the maximal interpretation, each transition represents an option, and the more options, the *larger* the extension.[2]

We will refer to this relationship between a process class and its extension as the *extension semantics* of the particular process representation.

The key point here is that what counts as a specialization of a given process model depends critically on what extension semantics have been assigned to that model. From this perspective we can see that this matter is not dealt with explicitly in the semantics of most process representations.[3]

---

[2] One can also imagine approaches that are more elaborate than either of these methods, but these suffice for the current analysis.

[3] As will become apparent when we contrast our approach with that of Nierstrasz (1993), there is room for interpretation in this regard even in the apparently straightforward case of state diagrams.

This lack of extension semantics introduces an ambiguity into attempts to specialize and classify processes, an ambiguity with important consequences for attempts to redesign and reuse process models, as discussed in Section 5 below.

It follows then, that in carrying out our analysis of state diagrams and dataflow diagrams, we will need to adopt some kind of extension semantics. In the interests of simplicity, throughout this paper we will use the "maximal execution set" approach described above. While this choice may not be optimal for many practical situations, it is ideal for our purposes, in that it highlights the potential difficulties which must be addressed in a consistent approach to process specialization.

2.2. Maximal Execution Set Semantics

Under maximal execution set semantics, each process model is understood as defining the universe of behaviors from which any process instance is to be constructed. This semantics seems especially well suited to circumstances in which it is more important to prevent undesirable consequences than to allow for creative elaboration because the system is not allowed to have any behavior outside the specified set (consider, for example, the case of modeling the operations of a nuclear reactor or intensive care unit). This movement from an all inclusive general case to more restricted special cases may also provide valuable support to the system designer by offering an explicit set of variations to choose from rather than an open-ended space of unspecified possible extensions (as would be the case with minimal execution set semantics).

Given this choice of extension semantics, we can describe specialization in terms of the maximal execution sets themselves:

Proposition: Given processes P and P` defined under maximal execution set semantics, with $S_P$ the maximal execution set for P and $S_{P`}$ the maximal execution set for P`, then P` is a specialization of P if and only if $S_{P`}$ is a subset of S_P.

Proof:[4] Appendix A

Having specified an execution set semantics and derived its implications for specialization, we now address the frame of reference used to describe a process and develop criteria for comparing processes with different frames of reference. This is critical to our treatment of activity decomposition which is an important feature of many process representations. Having completed this analysis we then introduce the notion of *specializing transformation*.

2.3. Frame of Reference

A process is among other things a set of possible behaviors, which we have been referring to as the *execution set* of a process. Note that any description of an execution set is made with respect to some *frame of reference* for the system: the frame of reference corresponding to the collection of attributes used to describe the set of possible behaviors which constitute that process. As we shall see it is possible to develop equivalent descriptions of a process (and its execution set) in a number of different frames of reference. In particular, we will introduce the notion of "refinement," which denotes a change to a "finer grained" frame of reference.[5]

For example, if the system of interest is an object moving in space, one might begin with a frame of reference with attributes for the position, mass, and velocity of the object, and then refine the frame of reference either by adding a new attribute, for example the temperature of the object, or refining an existing attribute, for example measuring position to the nearest meter as

---

[4] For briefer exposition, all the proofs are presented in the appendices.

opposed to the nearest kilometer.[6] To fully develop our approach to specializing transformation, we will need to integrate this notion of refinement into our view of specialization:

We have shown that specialization can be viewed as a restriction on the maximal execution set of a process: a process p1 is a *specialization* of a process p0 if its maximal execution set is a subset of the maximal execution set of p0. This result must now be restated to take into account frame of reference; there are two cases to consider:

1. *Both processes are described using the same frame of reference*. In this case the maximal execution sets of the processes are described in the same terms and can be compared directly. Thus p1 is a specialization of p0 if and only if the maximal execution set of p1 as described using the given frame of reference is a subset of the maximal execution set of p0 as similarly described.

2. *The processes are described using different frames of reference, but there exists a "common" frame of reference (which is a refinement of both of these)*.[7] In this case, p1 is a specialization of p0 if and only if the refinement of p1 is a specialization of the refinement of p0 under the common frame of reference. Thus this second case is reduced to the first by means of refinement.

2.4. Specializing Transformations

We propose that one useful way to operationalize this notion of specialization is in terms of a set of transformations for any particular process representation, which, when applied to a process

---

[5] The discussion which follows is in the spirit of the treatment of refinement and abstraction given by Horning and Randell (1973), who provide a lucid and wide-ranging exploration of this topic.

[6] A more formal definition of refinement is given in Appendix B and is employed in deriving further results below.

[7] Note that the common frame of reference may be identical to one of the given frames.

description, produce a description of a specialization of that process. The two-part definition of specialization given above suggests that two sorts of transformations will be needed:

A *specializing transformation* is an operation which, when applied to a process described using a given representation and a given frame of reference, results in a new process description under that representation and frame of reference corresponding to a specialization of the original process. Specializing transformations change the extension of a process while preserving the frame of reference.

In contrast, a *refining transformation* is an operation which changes the frame of reference of a process while preserving its extension, producing a process description of the same process under a different frame of reference.

For each type of transformation there is a related inverse type: a *generalizing transformation* acts on a process description to produce a generalization of the original process and is thus the inverse of a specializing transformation. Similarly, an *abstracting transformation* is the inverse of the refining transformation, producing a new description of the same process under a frame of reference for which the original frame is a refinement.

Given that the refining/abstracting transformations preserve the extension of a process, it follows from our definition of process specialization that a specializing transformation composed with refining/abstracting transformations in any sequence produces a specialization. The analogous statement holds for generalizing transformations.

A set of refining/abstracting transformations is said to be *complete* if for any process p described under a frame of reference, the description of that process under any other frame of reference can be obtained by applying to p a finite number of transformations drawn from the set.

A set of specializing transformations is said to be *locally complete* if for any frame of reference and any process p described using that frame of reference, any specialization of p

described under that frame of reference can be obtained by applying to p a finite number of transformations drawn from the set. Local completeness corresponds to the first part of the definition of process specialization given above.

There is also a notion of completeness corresponding to the second part of the definition. A set of specializing transformations and refining/abstracting transformations is said to be *globally complete* if for any process p, any specialization of p for which a common frame of reference exists can be obtained by applying to p a finite number of transformations drawn from the set.

Proposition: Let A be a complete set of refining/abstracting transformations and S be a locally complete set of specializing transformations. Then $A \cup S$ is globally complete.

Proof: Appendix C

3. STATE DIAGRAMS

The first example of process representation that we shall consider is the finite state machine or state diagram. State diagrams are often used to represent the dynamic behavior of systems. The circles in a state diagram correspond to states of the system being modeled, and the arcs connecting those circles correspond to the events that result in transitions between those states. The state diagram thus defines a set of possible sequences of events and states. Each state diagram must include at least one initial state (identified by a wedge, also known as an "initial state marker") and one final state (identified by a double circle, also known as a "final state marker"). All sequences must begin with an initial state and continue until they terminate with a final state. The set of states included in a state diagram can be thought of as a one-dimensional attribute space where the single attribute has values which correspond to the possible states. A system behavior corresponds to a sequence of these states and each state diagram defines a set of

such behaviors, which we interpret here as a maximal execution set. Under maximal execution set semantics, the process class described by this state diagram is taken to include all systems whose execution set is some subset of this maximal execution set. For example, the state diagram in Figure 2 permits the event sequences ac, abac, ababac, abababac, and so on. This entire set of sequences can be described by the regular expression a(ba)*c.

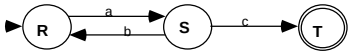

Figure 2. State diagram as a class of possible event sequences

Using the approach developed in Section 2, we can then define a state diagram D` to be a specialization of state diagram D if and only if either:

1. The set of sequences permitted by D` is a subset of the set of sequences permitted by D.
2. Either D or D` can be refined such that (1) holds. (This essentially amounts to resolving differences in the granularity of the two process descriptions by decomposing states into substates.)

We will first identify a complete set of refining/abstracting transformations. Then we will identify a set of specializing transformations which is locally complete. Global completeness of the union of these transformation sets then follows from the proposition given at the end of Section 2.

3.1. Refining/Abstracting Transformations for State Diagrams

Proposition: The following constitutes a complete set of refining/abstracting transformations for state diagrams:

Refinement by exhaustive decomposition. Replace a state by a mutually exclusive collectively exhaustive set of substates. Add events corresponding to all possible transitions between

substates. For each event associated with the original state, add a corresponding event for each of the substates.

Abstraction by total aggregation. If a set of states is completely interconnected by events and an identical set of "external" events is associated with each state in the set (in other words, if this set of states has the properties of an exhaustive decomposition as described above), replace that set of states by a single state which represents their aggregation. Associate with this state the same set of events which was associated with each of the substates.

The proof is found in Appendix D.

3.2. Specializing Transformations for State Diagrams

Proposition: The following constitutes a locally complete set of specializing transformations for state diagrams:

Delete an individual event. This removes a possible transition between events and thus the new diagram is specialized to exclude all behaviors that involve such a transition.

Delete a state and its associated events. The new diagram is specialized to exclude all behaviors that involve the deleted state.

Delete an initial state marker. This transformation is subject to the condition that at least one initial state marker remains. The new diagram is specialized to exclude all behaviors that begin with the affected state.

Delete a final state marker. This transformation is subject to the condition that at least one final state marker remains. The new diagram is specialized to exclude all behaviors that end with the affected state.

The proof is found in Appendix E.

It follows directly from the propositions proven so far that the union of the sets of transformations given above is globally complete.

Finally, while the preceding set of transformations is thus complete it may sometimes be convenient to employ other specializing transformations. In particular we will make use of the following transformation:

Specialize a state. Replace a state in the original state diagram by one of its substates. This transformation can be expressed in terms of the above set by first exhaustively decomposing a state into substates and then deleting all but one of them.

4. EXAMPLE: RESTAURANT INFORMATION SYSTEM

To better understand how the approach we have developed might be of practical value, we present the following stylized example involving a restaurant information system based loosely on the work of Salancik and Leblebici (1988). This example is chosen because of its relative simplicity, and because of the familiarity of the restaurant domain.

Imagine that you are a systems analyst charged with developing an information system to support the operational side of a large restaurant or chain of restaurants. You might include as part of your analysis a state diagram representing the flow of events involved in a "meal transaction" in a restaurant. That is, the flow of events involved in the delivery of meals to customers and the collection of payment.

We will assume that based on interviews and observations you have determined that any meal transaction will be composed of the following set of five activities: ordering a meal, cooking, serving, eating, and paying. Furthermore your interviews suggest that in the restaurants in

question these steps always occur in a single sequence, leading to the simple state machine depicted in Figure 3.[8]

4.1. Building a Specialization Hierarchy

Having successfully developed software to support the operations of this first group of restaurants, you are called upon to modify the software to work in three other food service environments: a fast food restaurant, a buffet, and a church supper. Based on further interviews and analysis you develop the state diagrams shown in Figure 4 to describe each of these processes.

Having observed that none of the four state diagrams developed so far is a specialization of any other, you apply the generalizing transformations to generate a generic restaurant process for which each of the above state diagrams is a specialization. As the diagrams differ only in the events they include, generalizing is simply a matter of adding each of the events from the other diagrams to the original diagram. The resulting diagram is shown in Figure 5.

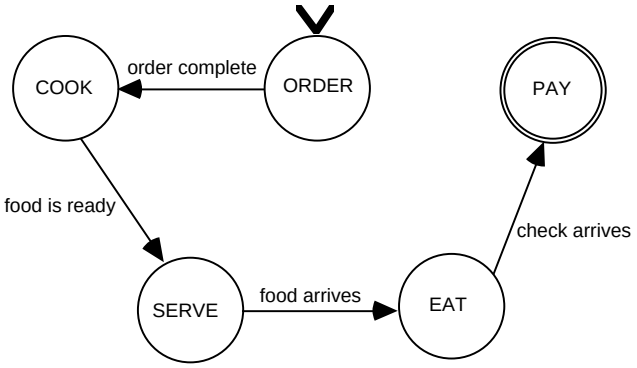

Figure 3. State diagram for full service restaurant

[8] While we have labeled the events in Figure 3, in general we will leave them unlabeled because the specific nature of the events is usually obvious, and in any case of limited relevance to the present analysis.

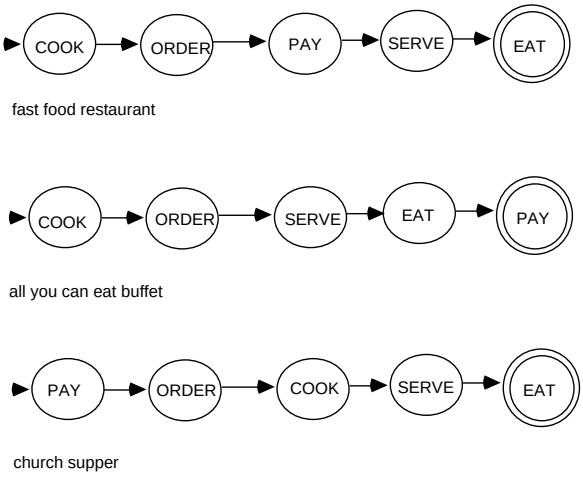

Figure 4. Additional restaurant state diagrams

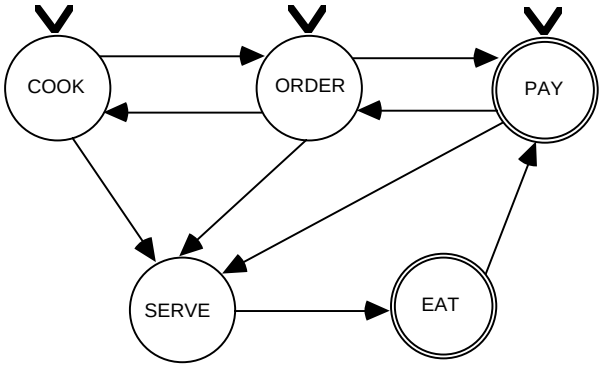

Figure 5. Generalized restaurant transaction

You have thus generated the specialization hierarchy depicted in Figure 6**.** Such hierarchies can contribute to software (and design) reuse by providing a taxonomy of previous designs that can be searched easily.

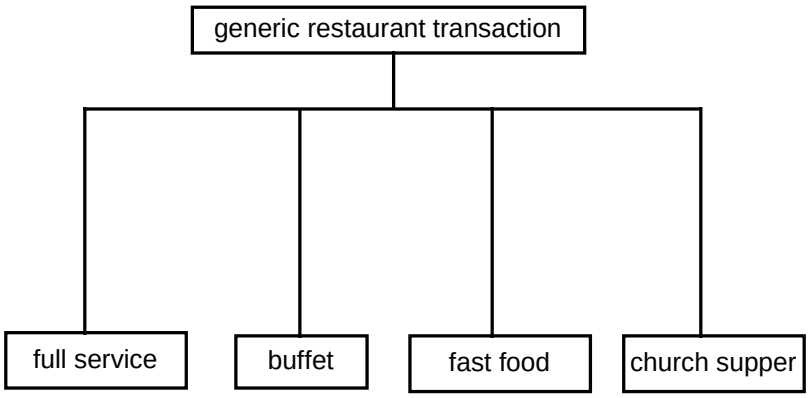

Figure 6. Initial specialization hierarchy for restaurant information system

4.2. Generating New Processes

Of special interest is the fact that design knowledge propagates up this hierarchy to the most generic diagram which thus contains accumulated knowledge about all variants of the restaurant process. This generic diagram can then be used to generate additional diagrams.

For example, imagine that you are now called upon to develop a specification to support a restaurant with both table service and a buffet. You can obtain a state diagram for such a hybrid by applying a series of specializing transformations to the generic diagram (see Figure 7).

To the extent that one is choosing among a set of preexisting functions resident in the most generic diagram, a story about the great artist Michaelangelo would seem to be relevant. Michaelangelo was asked how it was that he was able to produce the extraordinary sculpture of David for which he is famous. He replied that he began with a block of marble and simply removed all the pieces that were not David, until only David remained. So with the most generic state diagram in a hierarchy, many diagrams can be generated simply by removing states and events that don't apply.[9]

---

[9] One hesitates to draw such a presumptuous comparison, but we can all aspire to produce systems with the grace, beauty, and intrinsic value exemplified by Michaelangelo's work!

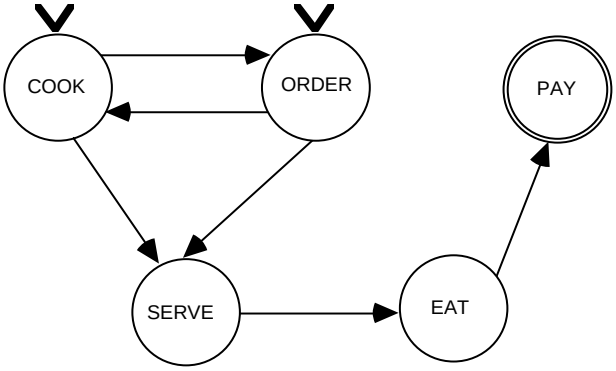

Figure 7. Full service restaurant with buffet

5. DATAFLOW DIAGRAMS

Having explored specialization of state diagrams in some detail, we now turn to dataflow diagrams. Dataflow diagrams are intended to show the *functionality* of a system: the various processes, and the flows of information and material which link them to each other, to inventories (data stores), and to various agents external to the system. A dataflow diagram (DFD) consists of a collection of processes, stores, and terminators linked by flows. A simple example taken from Yourdon (1989, p.141) is given in Figure 8. This discussion follows the approach taken by Yourdon (1989, chapter 9), to which the interested reader is directed for a more detailed exposition.

Processes, shown as circles in the DFD, are the component actions or subprocesses which together constitute the overall process or system being represented in the diagram. *Stores*, represented by pairs of parallel lines in the DFD, are repositories of the data or material carried in the flows. *Terminators*, shown as rectangles in the DFD, represent the actors, external to the system being modeled, which interact with the various system processes. *Flows*, shown as arrows in the DFD, represent the movement of information or material between processes, terminators, and stores.

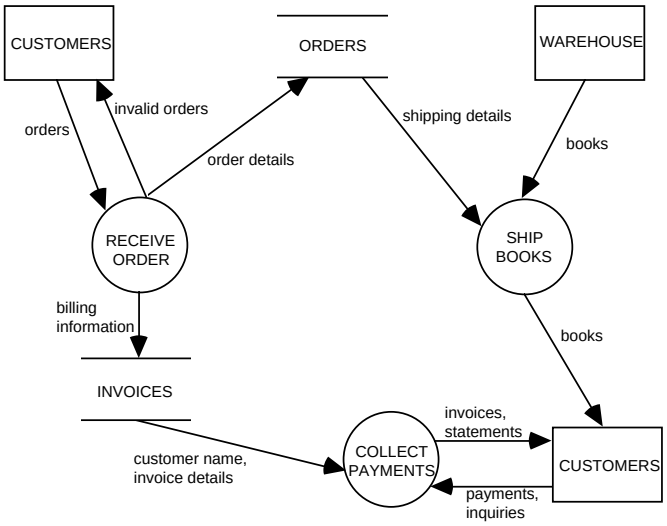

Figure 8. Example of a dataflow diagram: order processing

5.1. Specialization of Dataflow Diagrams

Before discussing specialization of dataflow diagrams, we must be more precise about the set of behaviors described by a dataflow diagram. While the DFD approach as usually presented does not specify what such a "DFD behavior " would look like, it seems reasonable to describe it as a sequence of processes and flows.[10] An immediate consequence of this approach under maximal execution set semantics is that executions of a particular DFD may only include processes and flows contained in that DFD. Note that terminators and stores are implicitly included in executions as the endpoints of flows.

A dataflow diagram does more however than simply list what processes and flows may occur in an instance. It also says something about the relationship between those flows and processes. For example, for each instance of a process which occurs in a DFD instance, one would expect some and possibly all of the flows into and out of that process to also occur.

[10] This discussion omits many aspects of DFDs that are worthy of attention, such as duration and sequencing of processes and flows.

However, in attempting to state these constraints precisely one must take a position on certain questions about how a DFD is to be interpreted. For example, in the present discussion we will assume that in general all flows into or out of a store or terminator may occur independently of each other.[11] We will also assume that each process instance must be accompanied by *at least* one inflow and one outflow, but that (without extending the dataflow representation) one cannot in general say more about which flows accompany a process execution without appealing to the semantics of the domain being modeled. For example, in Figure 8, any instance of Ship Books must involve all three flows: an incoming shipping memo and books, which are transformed into an outgoing shipment to the customer. However, in the same diagram the flow of an order into Receive Order may result in a flow of order details into the Orders store or the flow of an invalid order back to the customer, but (presumably) not both. This latter issue appears to represent a fundamental ambiguity in the dataflow representation: it would seem that there is no domain independent interpretation of a DFD which permits a consistent definition of its class membership.

Since we have no domain independent interpretation of a DFD as defining a class, specialization cannot be extended to DFDs in a domain independent fashion. That is, in general one cannot determine whether one DFD is a valid specialization of another without explicating which flows are mandatory and which are optional and under what circumstances, information which is not captured in the DFD itself.

These ambiguities in the dataflow diagramming technique are well-known and resolutions have been proposed (France 1992). We can proceed without such extensions, however, by

---

[11] For example, one might have a store of customer information that is updated by one process and queried by another, with the two processes and their flows occurring asynchronously. One can, of course, easily imagine dataflow diagrams in which the semantics require synchronization of flows into and out of a data store, and in these situations the approach we are taking is somewhat lax, permitting sequences

limiting ourselves to transformations which neither add flows to nor delete flows from a process component. These transformations will then be specializing under any interpretation of process flows, because such flows are left intact under the transformation. Interestingly, even under this constraint we obtain a set of transformations which is rich enough to be useful, as will be illustrated in Section 6 below.

We are now in a position to specify what executions are in the maximal execution set of a dataflow diagram. We can then identify transformations which result in a restriction on the maximal execution set and thus (as argued in Section 2 above) result in a specialization. The maximal execution set of a dataflow diagram includes all sequences of processes and flows which satisfy the following constraints:

All processes and flows in the sequence appear as components of the dataflow diagram.

Each input flow or output flow to a process which appears in the sequence must be associated with at least one instance of that process in the sequence.

Each process which appears in the sequence must have at least one associated input flow and one associated output flow.

We can now give a definition of specialization for dataflow diagrams which follows directly from Section 2. That is, we can then define a dataflow diagram D` to be a specialization of dataflow diagram D if and only if either:

1. The set of sequences permitted by D` is a subset of the set of sequences permitted by D.

2. Either D or D` can be refined such that (1) holds. (This essentially amounts to resolving differences in the granularity of the two process descriptions by decomposing process components.)

---

that should be prohibited. While addressing this issue is beyond the scope of the present paper, it is worth noting that this problem might be resolved by introducing additional constructs to indicate the presence of such synchronous flows.

Having defined the relationship between a dataflow diagram and its execution set in terms of the constraints above, we are now in a position to identify a set of specializing transformations which operationalize the above definition. For this it will be useful to first introduce a set of refining/abstracting transformations which in turn requires a formal definition of the dataflow diagram and its attribute space. The formal definitions and analysis are given in Appendices F and G. In the discussion which follows we will simply summarize and briefly motivate the results.

5.2. Specializing and Refining Transformations for Dataflow Diagrams

For purposes of the current analysis of dataflow diagrams, we need only consider a single refinement -- *exhaustive process decomposition* -- and its corresponding abstraction -- *total process aggregation*. Intuitively we achieve exhaustive process decomposition by replacing a component process with a set of subprocesses (including a generic process so that the decomposition is exhaustive) interconnected by all possible generic flows and with a copy of each "external" input and output flow linked in turn to each of the subprocesses. The presence of all possible flows and the generic process insures that the decomposed process represents a true refinement (i.e. does not restrict the extension of the original dataflow diagram in any way). In practice, of course, decomposition of processes in dataflow diagrams does not include all possible flows and subprocesses for such decomposition involves both a refinement and a specialization (restriction of extension) of the original dataflow diagram (which is also consistent with other decompositions). The exhaustive process decomposition is thus of primarily theoretical interest: a kind of "refinement benchmark" against which specializations which involve decomposition can be analyzed.

As noted above, in developing a set of specializing transformations we will limit ourselves to transformations which preserve flows in and out of processes. Note that any such transformation must be specializing, because any executions in the maximal execution set of the resulting dataflow diagram must satisfy the MES conditions for the original dataflow diagram as well, since these only involve the relationship between flows and their associated processes, and these are not affected if we preserve flows in and out of processes. We can identify several useful specializing transformations which are consistent with this constraint:

Deletion[12] *of a connected collection of components whose bordering components in the original diagram are all either terminators or stores.* To get the sense of this transformation, imagine the original DFD as a physical structure constructed by bonding the various components together, and further imagine that these bonds are unbreakable with the exception of any bond between a flow and either a terminator or store. By breaking these latter bonds, one may in some cases be able to separate the diagram into several pieces. This transformation consists of removing a single one of these pieces while leaving the rest of the diagram and all its bonds intact (e.g. Figure 11, Figure 12, and Figure 13 in Section 6). Note that this transformation preserves the constraint on flows since all remaining processes have all their flows intact (the only components with deleted flows are terminators and stores). The intuitive justification for this transformation is that we take stores and terminators to be asynchronous and thus an execution may be restricted to one side or the other of a boundary defined by these components.

Decomposition of a process. Any process in a DFD can be decomposed into a lower level DFD, as long as the flows into and out of the decomposition are consistent with the flows in the top

---

[12] The notion that a diagram can be specialized by deleting (rather than adding) a component may seem at odds with our common understanding that an object is specialized by adding (or subtyping) an attribute. This apparent contradiction can, however, be resolved. The short version of the argument is that deletion of a process component is in this case analogous to the sub-typing of an attribute rather than the deletion of an attribute. (See Section 8 below for fuller discussion of this issue).

level diagram. Note that this kind of decomposition is not exhaustive in the sense of exhaustive process decomposition (which we argued above is a refinement). This "non-exhaustive" decomposition can be thought of as a refinement (exhaustive process decomposition) composed with a specialization (deleting some subprocesses and decomposed flows). Note that our constraint that flows associated with a process are preserved is satisfied by the "flow consistency" aspect of this form of decomposition. That is, we require that for each flow into or out of the decomposed process, there be at least one identical flow into or out of one of the resulting subprocesses.

Specialization of a component. If one specializes any component (terminator, store, process, or flow) of a dataflow diagram, the resulting diagram will be a specialization of the original diagram. Note that here again we preserve flows associated with each process. In particular, a specialized process can be thought of as a kind of subset of the original process, which is to say we replace the original process with a subprocess, which means that specialization of a process is nothing more than a kind of process decomposition. A similar argument might be made for the specialization of flows. Finally, under the semantics we are employing for dataflow diagrams, terminators and stores figure into the maximal execution set of a dataflow diagram only as endpoints of a flow, which is to say that they are essentially attributes of some flow, which means that specialization of terminators and stores is a kind of flow specialization, which as we have just noted, can be understood as a kind of flow decomposition.

6. AN EXAMPLE: GENERATING ORDER PROCESSING ALTERNATIVES FOR E-BUSINESS

Having established a method for systematically generating process specializations, how might we use this method to support process redesign? We illustrate the possibility here with an e-business design scenario: Consider a manager exploring possible changes to an order fulfillment process occasioned by a shift from a traditional brick and mortar enterprise to an e-business. For such a manager, a generic account of this process (such as that taken from Yourdon (1989) and depicted in Figure 10 below) is of potential value in that it identifies the key activities and flows to be addressed. However, the role played by these elements may change, up to and including the possibility that some of them may simply go away when order fulfillment moves onto the internet.

What we propose here is a procedure, based on the notion of process specialization, that this manager might employ to generate a set of process variants that call into question assumptions implicit in the generic order fulfillment process and which therefore support a systematic exploration of design possibilities for the new process. This procedure is illustrated in Figure 9, which can be read from top to bottom as a sequence of steps.

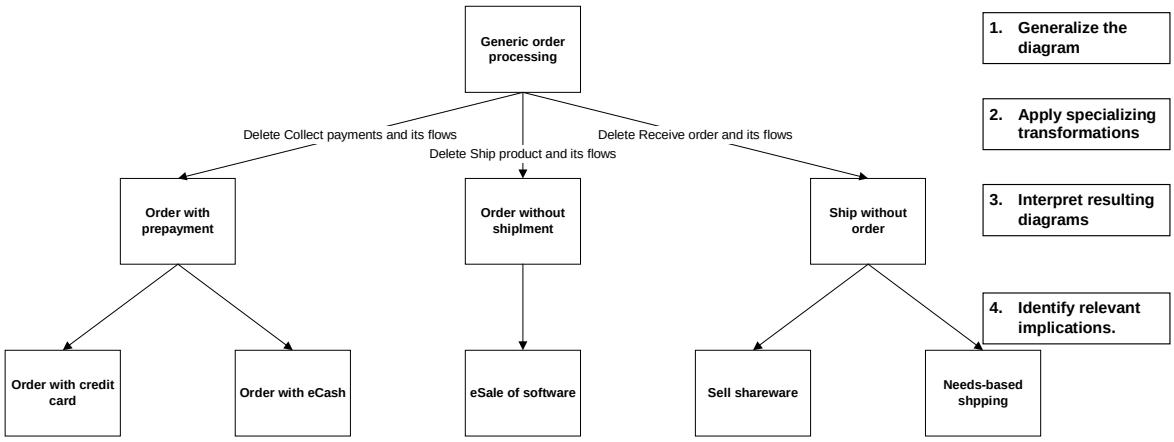

Figure 9. Taxonomy of Order Processes

1. First, a suitably generic representation of order processing must be obtained. For purposes of our example, we begin with the Yourdon diagram and generalize it in a manner consistent with how we have defined specialization for dataflow diagrams: the process Ship Books is generalized to Ship Product, and the flows labeled "books" are generalized to product flows. The resulting generalization is depicted in Figure 10.

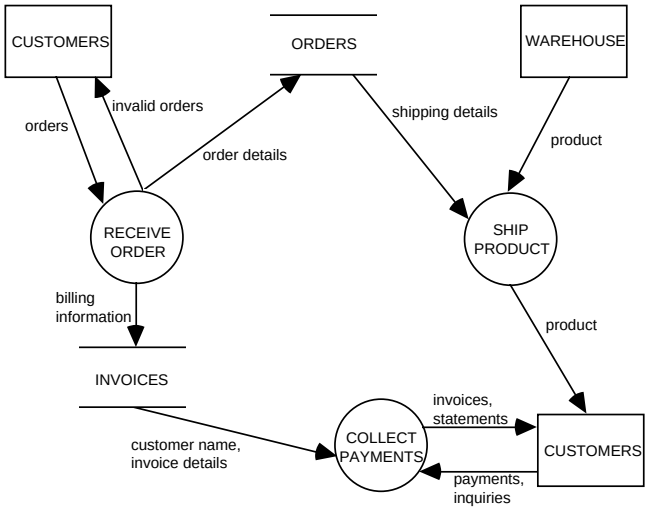

Figure 10. Order processing abstracted from books to products

2. Having established a starting point for her analysis, the manager then systematically applies one or more specializing transformations to the generalization in order to generate a set of alternatives. In this example we focus on the set of dataflow diagrams which are generated by deleting connected portions of the DFD which border on stores and terminators.[13]
3. Once these specializations are obtained, the next step is to find a meaningful interpretation for the resulting diagrams: how do assumptions need to change in order to make sense of each specialization as some kind of order fulfillment process? Note that it is possible that several interpretations will arise (in which case they should all be included), or that no

[13] Note that the original DFD consists of three connected groups of components joined by the two stores Orders and Invoices, and the terminator Customers. There are thus six possible specializations which result from deleting one or more of these groups from the diagram: three

plausible specialization arises. In this latter case one might then consider whether the proposed specialization violates some implicit constraint (a realization which is no doubt useful in itself). In the absence of such an "impossibility argument," one might want to retain the specialization against the future possibility of a plausible interpretation. For example, this may provide a framework for identifying new organizational forms as they emerge in the future(by understanding them as instances of a previously hypothetical specialization).

4. Finally one must consider the relevance of each specialization to the problem at hand, in this case the transition to electronic commerce.

What follows is a brief discussion of the specializations which result from this procedure.

Order processing with pre-payment. Figure 11 depicts a specialization of the original DFD in which the Collect Payments process and its associated flows have been deleted. Note that the flow of orders has been specialized as well to indicate that cash must accompany each order. In this specialization any order without accompanying payment is returned to the customer as invalid, otherwise the order is forwarded to the Ship Product process and the invoice information is stored in the Paid Invoices store (a specialization of the original Invoices store which reflects

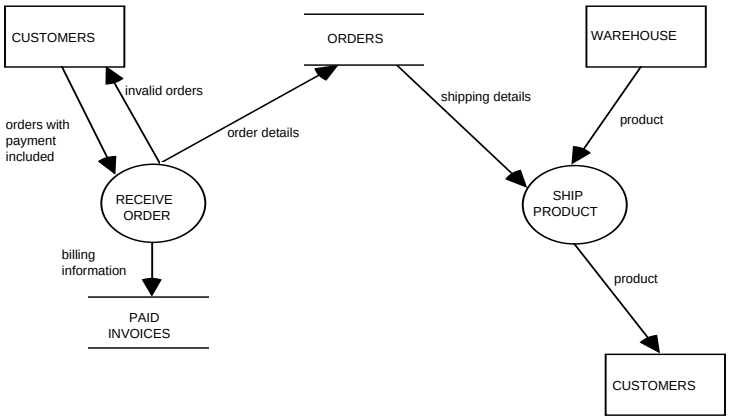

Figure 11. Order Processing with Pre-Payment

---

specializations in which two of the groups are deleted, and three specializations in which one of the groups is deleted. In this example we will restrict ourselves to the "less radical" transformations in which only one of the groups is deleted.

the lack of unpaid invoices in this system). One example of this alternative is the common form of order processing for eBusiness companies, the payment with credit card accompanies the order. Other examples include the uses of eCash or gift certificates, which may require simple accounting adjustments for the payment. These examples constitute further specializations of this alternative.

Order processing without shipment. Figure 12 depicts the diagram which was specialized from the original DFD by deleting the Ship Product process and specializing its associated flows and the various stores and flows appropriately. This specialization would be possible to implement when there is a way for customers to obtain products without the company shipping them. For example, software products can be made available over the net for the customers to download, as with the company software.net, while the payment can follow later.

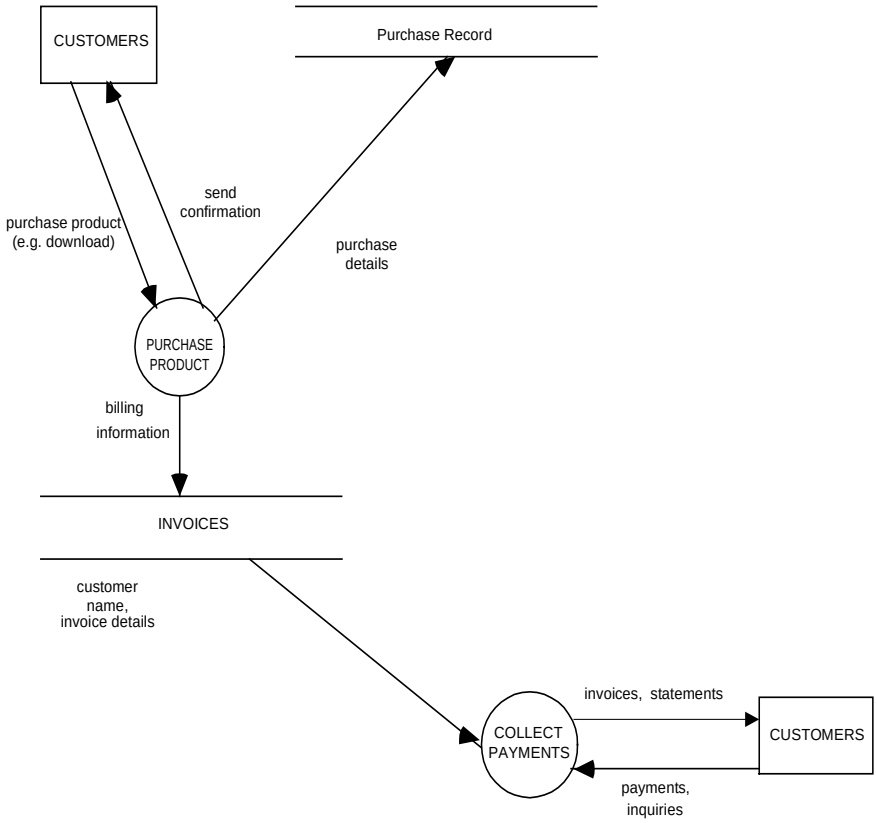

Figure 12. Order Processing without Shipment

Order processing without order. Figure 13 depicts a specialization of the original DFD in which the Receive Order process and its associated flows have been deleted. This diagram might be interpreted as depicting a process in which products are shipped, unasked for, to prospects who are then billed for the products. Although this practice sounds unscrupulous, there do appear to be acceptable instances of this process as, for example, when "shareware" is shipped with a computer system or book, along with an electronic invoice that the recipient has no obligation to pay but can pay if he or she likes the product. Another example is when a company can detect your need and automatically ships the products to fulfill the need. Although it sounds somewhat futuristic, some companies come close to implementing this alternative. For example, the eCompany, streamline.com, manages your household food inventory for you and delivers what you need without you having to ask for it.

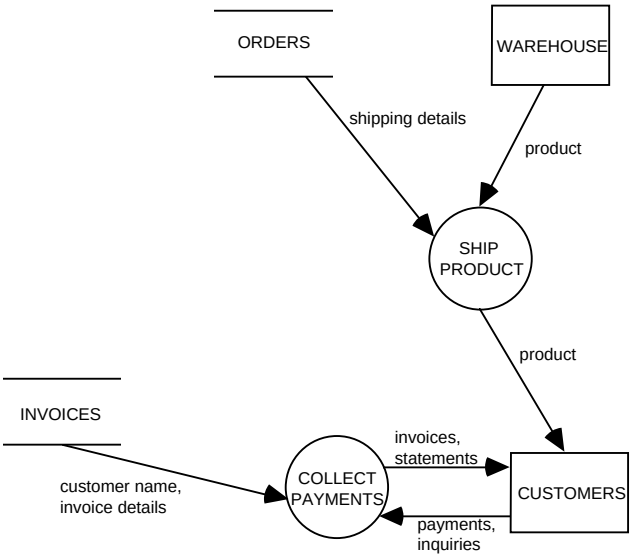

Figure 13. Order Processing without Order

This analysis of specialization in dataflow diagrams can be summarized by the specialization/generalization hierarchy given at the outset of this section (Figure 9). Such a hierarchy provides both a process taxonomy and a structure which facilitates the systematic

consideration and reuse of alternative designs. Thus, one can enter the hierarchy with a particular process, "move up" to a more abstract process, and then consider not only plausible alternative process designs, but also identify other processes which might serve as sources of inspiration (Malone et al. 1999).The above analysis demonstrates that the use of specializing transformations, even when they are applied to a commonly understood business process such as order processing, can help us explore organizational alternatives.

7. RELATED WORK

The notion that processes can be specialized is not new. An informal notion of process specialization can be found in a number of organizational design approaches. Checkland (1981) builds the design process on an idealized abstract model (referred to as a "root definition") which is then adapted and interpreted to the situation at hand. Senge (1990) introduces the notion of systems archetypes, which provide a generalized vocabulary that can be adapted to particular organizational situations. The practice of benchmarking can also be viewed as constructing idealized models of best practice which are to be adapted to specific organizations. Our approach can be viewed as formalizing the intuitions behind these studies.

More formal notions of process specialization can be found in the studies of AI planning (Friedland and Iwasaki 1985, Stefik 1981), conceptual modeling (Borgida et al. 1993), object-oriented design (Nierstraz 1993)[14], and workflow (van der Aalst and Basten 1999). In all of these studies, however, process specialization is defined for the specific representation being proposed and none of these studies are concerned with process design.

---

[14] Interestingly, Nierstrasz operationalizes specialization for state machines in a manner almost diametrically opposed to our own. As we discuss in section 8 below, however, this apparent inconsistency can be resolved as a difference in how the state machines are interpreted as objects -- what we refer to in section 2.1 as the distinction between *maximal* and *minimal execution semantics.*

Keller & Teufel (1998), in their approach to SAP customization, propose a series of transformations (e.g. selection of relevant modules, deletion of extraneous functionality, and setting of parameters) to be applied to the SAP code base. These transformations, when formalized, can be viewed as a precursor to our notion of specializing transformation. However, this work makes no attempt to extend the notion of process specialization beyond the SAP context.

There are other studies of process representation which take approaches similar to the one we propose. For example, the process formalism in this paper and the definition of refinement in particular are somewhat similar to the approach taken by Horning and Randell (1973). Horning and Randell also develop a notion of one process *containing* another, which is equivalent to our definition of process specialization. This concept, however, plays a secondary role in their analysis. The F-ORM method, a method for specifying applications, discusses "transformation operations" among process representations (De Antonellis et al. 1991). In this method, however, these transformations pertain to the decomposition relation, not specialization.

What is unique in our approach then, is the notion that process specialization can be defined in a way that permits it to be incorporated into existing process representations by means of a set of specializing transformations. These transformations, when applied to a process in a given representation, result in a specialization of the original process. It is this transformational aspect of our approach which we claim provides a kind of generativity which will prove invaluable to process redesign. By supporting specialization for existing process representations we hope to make this approach available as an extension to existing systems analysis and design methods.

As mentioned at the outset, the suggestion by Malone et al. (1999) that specialization can be applied to processes has been the foundation for this research effort. In their process handbook, Malone et al. arrange business processes in a specialization hierarchy. Entries in this

"handbook" can be viewed at various levels of decomposition. One can also traverse the specialization hierarchy to identify interesting variants of a given process, including specializations, generalizations, and "siblings" (alternative specializations of the process's parent in the hierarchy).

Some modeling languages such as TAXIS (Borgida et al. 1993) define a kind of specialization specific to their process representation. Our goal, on the other hand, is to define a general method for specializing processes under any process representation (even one that does not build the notion of specialization into its semantics, such as the state diagram or dataflow diagram).

As we have emphasized throughout this paper, the most obvious point of comparison for our work is the specialization of objects. Indeed, the research most closely related to our own would appear to be Nierstrasz's work on defining a subtyping relationship for active objects (Nierstrasz 1993). Nierstrasz treats an object as a finite state machine which defines the communications protocol supported by that object: the messages it accepts, the services it provides. He then defines a subtyping relationship on these state machines. This would appear to be quite similar to our own efforts to define a specialization relationship for state machines.

In comparing our work with Nierstrasz, four differences are salient:

1. Nierstrasz is concerned with state machines as defining the interactions between individual objects, whereas we are concerned with state machines as describing the internal behavior of entire systems.
2. Nierstrasz provides an algorithm for *determining* whether the subtype relationship holds between any two state machines. We provide a set of transformations for *generating* specializations and generalizations from a state machine. These two operationalizations would appear to be complementary; one can imagine situations in which it would be desirable to have the capability both to analyze and generate specializations.

3. Nierstrasz is concerned exclusively with state machines, where we are interested in applying our approach to a number of process representations.
4. Probably most intriguing is that our definition of specialization is almost exactly the opposite of Nierstrasz's definition of subtyping. For Nierstrasz, a state machine which is a subtype must accept a *superset* of the sequences accepted by the supertype, otherwise it cannot be substituted safely for that supertype (Nierstrasz 1993). For us, a state machine which is a specialization must accept a *subset* of the sequences accepted by its generalization, otherwise it will not represent a restriction in extension.

This apparent contradiction is easily resolved however by observing that Nierstrasz implicitly employs a minimal execution set semantics instead of the maximal execution set approach taken here. Given minimal execution set semantics, clearly a specialization must support a superset of the original minimal execution set and thus a state diagram is specialized by *adding* states and events. As shown above, our choice of semantics leads to exactly the opposite result.

While this dramatic divergence can thus be explained by a difference in semantics, the question remains whether an approach that involves specialization by deletion can be reconciled with the standard object-oriented framework. In section 8 below, we argue that, despite surface differences, our approach is entirely consistent with the object-oriented approach to specialization: what appear to be significant inconsistencies between the two types of specialization disappear when one looks more fully into the matter.

8. ARE THERE TWO KINDS OF SPECIALIZATION?

In reconciling our approach with object specialization, there are two salient issues to be addressed. The first, raised in the Nierstrasz comparison above, is whether it is proper to

specialize a process by deleting components, given that objects are specialized by doing what appears to be exactly the opposite: adding attributes.

The second issue follows as an implication of the first: if we specialize by deletion, then adding an attribute to a process requires adding the same attribute to its parents in the specialization hierarchy. Thus process modifications may propagate upwards, again in apparent contradiction of the object-oriented approach, where changes propagate downwards in the specialization hierarchy.

We deal with these two issues in turn:

8.1. Issue: Specialization by deletion

Deleting attributes when specializing is generally not permitted because, among other things, it violates the principle of substitution, in that the specialized object cannot be universally substituted for the original because references to the missing attribute may result in error. Under maximal execution set semantics, however, process specialization appears to make extensive use of this forbidden "specialization by deletion." For example, many of the specializing transformations for state diagrams described above involve deleting parts of the diagram. A closer examination, however, reveals that this is not a case of deleting attributes: when deleting parts of a state diagram one is altering a representation of the process but the things deleted are not themselves attributes of the process. In other words, one should not confuse the *maximal execution set* of a process *description* with the *execution set* of a process *instance*.

One way to see this is to note that object specialization involves subtyping one or more attributes, and subtyping is in a sense a kind of deletion: one makes the type of the attribute more restrictive and thus makes the set of permissible values smaller (which is kind of like deleting

from a list of permissible values). If one represents the type of an attribute graphically, this subtyping may be manifested as deleting elements from the graphical representation.

To take a simple example, consider a numeric attribute with type: INTEGER IN 1-10. One might choose to represent this type as a list of allowable numbers. Thus the type would be represented as

1 2 3 4 5 6 7 8 9 10

If one specializes by subtyping to INTEGER IN 4-7 one must delete elements from the representation to obtain:

4 5 6 7

This may appear to be "specialization by deletion" if one focuses on the representation, but clearly it is simply specialization by subtyping.

While this example may appear to be contrived, it is exactly analogous to the state diagram example. Deleting transitions in a state diagram corresponds to subtyping an attribute of the corresponding process. More specifically, the execution set of a process instance can be viewed as the value of an "executions" attribute. The maximal execution set of the process class is then the type of this attribute. As it turns out, one can represent this type as a collection of ordered pairs of states which are equivalent to the state machine depicted in the state diagram. One can then show that subtyping this attribute corresponds to deleting events or states from the state diagram as follows:

Observe that under maximal execution set semantics, an instance of the process described by a state diagram is a system which realizes some subset of that diagram's maximal execution set. Then the *actual* execution set of this instance is an attribute of the process. The value of this *ExecutionSet* attribute is then a set whose elements are of type *execution*. An element e of type execution is in turn defined as an ordered tuple of length $n > 1$, such that: for any i, $1 \leq i \leq n$, if a

and b are the ith and (i+1)th components of e respectively, then <a,b> ε E, where E is the set of all events in the state diagram, represented as ordered pairs of states. Note, then that we can subtype execution by restricting the scope of E to some proper subset. Note that this restriction involves removing ordered pairs from E which is equivalent to deleting these events from the state diagram. Thus the state diagram is a graphical representation of the execution type of the ExecutionSet attribute of a process, and deleting events is a form of subtyping.

Thus "specialization by deletion" can be seen to actually be specialization by subtyping.

8.2. Issue: Upward propagation vs. downward propagation

In object specialization hierarchies, the inheritance of attributes flows downward, but changes at the leaves of a *process* hierarchy seem to propagate upwards. In fact we will argue that this upward propagation can occur in any specialization hierarchy, and that this phenomenon is of potential interest to the systems designer.

Consider what happens to an object in a specialization hierarchy when one changes the attributes of one of its specializations (i.e. one of its children). Clearly adding additional attributes or further subtyping of attributes simply further specializes the child object and has no effect on the parent object. However, if one for some reason needs to "supertype" an attribute of the specialization (for example, one is validating the object model and discovers that the type of an attribute of some object is overly restrictive), a conflict may be introduced into the specialization hierarchy if the new type of this attribute is no longer a subtype of the corresponding attribute in the parent process. Then the child process is no longer a specialization.

If one is to be strict about specialization and it turns out that the new change in type is unavoidable, then one would have to resolve the situation by modifying the type of the parent

object in a similar fashion. This would follow logically given that the child is a specialization of the parent and the type of the child is now correct. Then, by definition of specialization, the type of the parent must be at least as inclusive. Now one has modified an attribute in the parent by supertyping and the same issue may arise with *its* parent with the result that a change in a leaf may necessitate changes in one or more ancestors, possibly all the way to the root of the tree.

Thus we can see that upward propagation is at least a theoretical possibility in any specialization hierarchy. It is important to note that such upward propagation is not normally supported in implementations of object oriented languages and would have to be carried out manually by a series of edits to the class definitions.

It is also important to note that upward propagation occurs only when one takes a strict approach to specialization, that is, requiring that the attributes of a specialization always be identical to or subtypes of the original attributes. If this strict approach is not enforced, then in the scenario above one would be free to add an inconsistent specialization without changing the attributes of the parent, and upward propagation would not occur. One would, of course, still be free to choose to modify the parent to reflect insights gained from developing the specialization, but there would be no requirement that such modifications be made.

The benefits of downward propagation (inheritance) are well known, and include the ability to define a new object incrementally by specifying only those aspects of the object which have changed, thus inheriting all the design knowledge associated with the parent node in the specialization hierarchy.

The benefits of upward propagation are those advanced by Malone et al. (Malone et al. 1999) when they suggest that a specialization hierarchy of processes will allow one to systematically identify a wide range of design alternatives. Upward propagation makes this possible by forcing all design knowledge upward from the leaves to the highest level of abstraction at which it is

relevant. Thus each process or object in the hierarchy reflects all the possibilities inherent in its descendants as illustrated with the restaurant and the order processing examples. This gathering of all possibilities can in turn lead to the other benefit mentioned by Malone et al.: generativity. For example, upward propagation may bring together a set of features originally present in distinct processes, which can then be recombined in unique ways by specialization, as for example, the different ways of providing a meal service or processing orders were generated from the respective generalized diagrams in the examples above.

9. CONCLUSIONS

We have explored how specialization can be applied to processes to take full advantage of the generative power of a specialization hierarchy. We have shown how specialization can be defined specifically for state diagrams and dataflow diagrams in the form of a set of transformations which, when applied to a particular dataflow diagram, result in process specialization. Furthermore, this method can be used to generate a taxonomy of processes to facilitate the exploration of design alternatives and the reuse of existing designs.

We have demonstrated that the rules by which a process diagram can be manipulated in order to produce a specialization (or a generalization) depend heavily on the semantics of the particular process representation. Thus the rules for specializing the state diagram differ in significant ways from those consistent with specialization of dataflow diagrams. Choose a different diagramming technique and you create a new hierarchy of diagrams that may offer additional insights.

The work presented is only a preliminary exploration of how the generative power of specialization hierarchies can be harnessed in support of organizational design. One natural

extension of this work is to explore additional process representations such as Petrie Nets and UML. One might also explore how the notion of specializing and generalizing transformations can be useful in other contexts, for example, the specializing of composite objects.

In summary, this paper has suggested that specialization, currently applied to great advantage in the modeling of objects, the nouns of the world, can be fruitfully applied to processes, the verbs of the world, as well. Together, specialization and abstraction give rise to a method of process analysis which shows promise as a means both for identifying new and interesting process possibilities and for gathering the multitude of alternatives so generated into process taxonomies, which can then serve as reservoirs of process knowledge (Malone et al. 1999).

ACKNOWLEDGMENTS

The authors would like to thank Paul Resnick for suggesting this topic and providing detailed comments, as well as Tom Malone for his feedback and encouragement. In writing this paper we have benefited greatly from discussions with Brian Pentland, Kevin Crowston, Chris Dellarocas, Fred Luconi, and Charley Osborn. Bob Halperin, Lorin Hitt, Kazuo Okamura, John Quimby, Ignascio Silva-Lepe, Tor Syvertsen, Marshall Van Alstyne, and the late Cheng Hian Goh also provided helpful comments. This research was sponsored by the Center for Coordination Science at the Massachusetts Institute of Technology and by the National Science Foundation (#IRI-9224093).

10. REFERENCES

Alexander, C. *The Timeless Way of Building*. New York: Oxford University Press, 1979.

Bansler, J. P. and K. Bodker "A Reappraisal of Structured Analysis: Design in an Organizational Context" *ACM Transaction on Information Systems*. 11(2) pp.165-193 1993

Booch, G. *Object Oriented Design with Applications*. Redwood City, California: Benjamin/Cummings. 1991

Borgida, A., J. Mylopoulos, and J.W. Schmidt "The TaxisDL Software Description Language, in Database Application Engineering with DAIDA" *Volume 1 of Research Reports ESPRIT,* M. Jarke, Editor. Springer-Verlag: Berlin. 1993 pp. 65-84.

Checkland, P. *Systems thinking, systems practice*. Chichester [Sussex] ; New York: J. Wiley, 1981.

Coad, P. and E. Yourdon *Object-Oriented Analysis*. Englewood Cliffs, New Jersey: Prentice-Hall. 1990

De Antonellis, V. , B. Pernici, and P. Samarati. "F-ORM METHOD: a F-ORM methodology for reusing specifications" in *Proc. IFIP TC8/WG8.1 Working Conference on the Object Oriented Approach in Information Systems*. Quebec City, Canada: North-Holland. 1991

De Champeaux, D. "Object-Oriented Analysis and Top-Down Software Development." in *Proc. ECOOP '91, European Conference on Object-Oriented Programming*. Geneva, Switzerland: Springer-Verlag. 1991

De Champeaux, D., L. Constantine, I. Jacobson, S. Mellor, P. Ward, and E. Yourdon. "Panel: Structured Analysis and Object Oriented Analysis" in *Proc. Conference on Object-Oriented Programming: Systems, Languages, and Applications / European Conference on Object-Oriented Programming*. Ottawa, Canada: Association for Computing Machinery. 1990

France, R.B. "Semantically Extended Data Flow Diagrams: A Formal Specification Tool." *IEEE Transactions on Software Engineering* **18**(4) 1992

Friedland, P. E. and Y. Iwasaki "The concept and implementation of skeletal plans." *J. Automated Reasoning* 1(2) pp. 161-208. 1985

Horning, J.J. and B. Randell "Process Structuring" *ACM Computing Surveys* **5**(1): p. 5-30. 1973

Keller, G. and Teufel, T. *SAP R/3 process-oriented implementation : iterative process prototyping*. Harlow, England ; Reading, Ma.: Addison Wesley Longman, 1998.

Krueger, C.W., "Software reuse" *ACM Computing Surveys* **24**(2): p. 131-183. 1992

Maksay, G. and Y. Pigneur. "Reconciling functional decomposition, conceptual modeling, modular design and object orientation for application development." in *IFIP TC8/WG8.1 Working Conference on the Object Oriented Approach in Information Systems*. Quebec City, Canada: North-Holland. 1991

Malone, T. W., K.Crowston, J. Lee, B. Pentland, et al. 1999 "Tools for inventing organizations: Toward a handbook of organizational processes" *Management Science* 45(3) pp.425-443

Nierstrasz, O. "Regular Types for Active Objects" in *Proc. Conference on Object-Oriented Programming: Systems, Languages, and Applications*. Washington, D.C.: Association for Computing Machinery. 1993

Rogers, R.V. "Understated Implications of Object-Oriented Simulation and Modeling" in *Proc. IEEE International Conference on Systems, Man, and Cybernetics*. Charlottesville, Virginia: IEEE. 1991

Rumbaugh, B., W. Premerlani, F. Eddy, and W. Lorensen *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall. 1991

Salancik, G.R. and H. Leblebici, "Variety and Form in Organizing Transactions: A Generative Grammar of Organization" in *Research in the Sociology of Organizations,* N. DiTomaso and S.B. Bacharach, Editor. Greenwich, Connecticut: JAI Press pp. 1-31 1988

Senge, P. . *The Fifth Discipline*. New York, Double Day/Currency. 1990

Stefik, M. "Planning with constraints (MOLGEN: PArt 1)." *Artificial Intelligence* 16(2): pp. 111-139. 1981

Taivalsaari, A. "On the notion of inheritance" *ACM Computing Surveys* 28(3) pp.438-479 1996

Takagaki, K. and Y. Wand. "An Object-Oriented Information Systems Model Based On Ontology" in *Proc. IFIP TC8/WG8.1 Working Conference on the Object Oriented Approach in Information Systems*. Quebec City, Canada: North-Holland 1991

van der Aalst, W. M. P. and Basten, T. "Inheritance of Workflows - An approach to tackling problems related to change." Eindhoven University of Technology Computing Science Reports #99/06 1999

Wegner, P. and S.B. Zdonik. "Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like" in *Proc. European Conference on Object-Oriented Programming*. Oslo, Norway: Springer-Verlag 1988

Yourdon, E. *Modern Structured Analysis* Yourdon Press Computing Series, Englewood Cliffs, New Jersey: Yourdon Press 1989

APPENDICES: FORMAL ANALYSIS

The following proofs and definitions are referenced in the main body of the paper.

Appendix A. Maximal Execution Set Semantics

Proposition: Given processes P and P` defined under maximal execution set semantics, with $S_P$ the maximal execution set for P and $S_{P`}$ the maximal execution set for P`, then P` is a specialization of P if and only if $S_{P`}$ is a subset of S_P.

Proof: If P` is a specialization of process class P, then for each behavior b ε $S_{P`}$, by definition of maximal execution set semantics, there is a process instance whose execution set contains b. This instance must also be in the extension of the more general class P and hence b ε S_P. It follows that $S_{P`} \subseteq S_P$. Conversely, let $S_{P`} \subseteq S_P$. Then consider any instance of P` with execution set e. By definition of maximal execution set semantics e $\subseteq S_{P`}$, hence e $\subseteq S_P$ and therefore p is an instance of P as well. It follows that all instances of P` are instances of P and thus P` is a specialization of P.

Appendix B. Refinement

The state of the system at any time is characterized in terms of some set of attributes which are ascribed to the system by an observer. The exact set of attributes may vary considerably from observer to observer and will reflect the abilities and interests of the observer, available technology, environmental conditions and so forth. The set of attributes employed in observing a system may be thought of as a frame of reference for that system, one of many possible such frames.

We assume that the set of attributes employed is fixed and finite and that each attribute can take on some set of possible values. We refer to this set of possible values as the range of that attribute.

We define an attribute space as the Cartesian product of the attribute ranges of all the attributes in a frame of reference. It follows that whenever the system is observed under that frame of reference its state will correspond to some point in the corresponding attribute space. Furthermore, each point in the attribute space corresponds to what may be a possible state of the system, although some of these points may refer to states which are in fact not realizable.

By behavior of a system we mean the evolution of that system's state over time, which is to say the path the system traces out in some attribute space. Thus any description of a system's behavior is made with respect to some frame of reference for that system.

Definition: an attribute space A` is a *refinement* of attribute space A if there is a surjective mapping M from A` onto A (i.e. the range of M includes every point in A; note that M maps the refinement into the original space rather than vice versa), with the property that a point a` in A` describes the state of the system if and only if M(a`) also applies. The intuition here is that if you refine your description of the system there are more possible state points, so that for each point in the original attribute space there is at least one point in the refined attribute space which is a description of the same state.

An attribute space A` is said to be a *strict refinement* if M is as above and is not also injective. That is, the inverse of M is not a function, or in other words, A is not also a refinement of A`. (This eliminates the trivial sense of refinement in which A and A` are essentially isomorphic.)

Given A`, a refinement of A, then a behavior b` in A` is said to be a *refinement of a behavior* b in A if the following conditions hold: 1. for every x` in b`, M(x`) is in b; 2. for every pair of points x1`, x2` in b`, if x1` precedes x2` in the path, then M(x1`) precedes M(x2`) or is identical to it. This last condition has to do with the fact that a path is a directed curve in attribute space and we need to make sure that the points are traced out in the same order in both curves. The idea here is that the refined version of the behavior maps point by point onto the original behavior. Note that given the finer grained view of a process which results from refinement, we must allow for the possibility that M(x1`) and M(x2`) are identical, and hence that several points in one curve may correspond to a single point in the other. Note, too, that it follows from this definition that every behavior in A will have at least one (and possibly more) refinements in A`.

Similarly, a process class p` in A` is said to be a *refinement of a process class* p in A, if for every behavior in the maximal execution set of p, all A`-refinements of that behavior are included in the maximal execution set of p`, and conversely, all behaviors in the maximal execution set of p` are A`-refinements of some behavior in the maximal execution set of p. Then it follows that every process represented as a maximal execution set in A will have exactly one refinement in A` and that this refinement is equivalent to the original process description in that both process descriptions lead to the same classification of behaviors.

Appendix C. Completeness of Specializing Transformations

Proposition: Let A be a complete set of refining/abstracting transformations and S be a locally complete set of specializing transformations. Then A ∪ S is globally complete.
Proof: Consider a process p0 and a specialization p1 for which a common frame of reference exists. Since A is complete, one can apply a finite sequence of transformations from A to p0 to produce its refinement in the common frame of reference. By local completeness one can then apply specializing transformations to produce the refinement of p1 (since it is a specialization of the refinement of p0 by assumption). Finally, by the completeness of A one can transform the refinement of p1 into p1. Thus there is a finite set of transformations from A ∪ S which produces p1 from p0.

Appendix D. State Diagrams: Refining Transformations

Proposition: The following constitutes a complete set of refining/abstracting transformations for state diagrams:

Refinement by exhaustive decomposition. Replace a state by a mutually exclusive collectively exhaustive set of substates. Add events corresponding to all possible transitions between substates. For each event associated with the original state, add a corresponding event for each of the substates.

Abstraction by total aggregation. If a set of states is completely interconnected by events and an identical set of "external" events is associated with each state in the set (in other words, if this set of states has the properties of an exhaustive decomposition as described above), replace that set of states by a single state which represents their aggregation. Associate with this state the same set of events which was associated with each of the substates.

Proof: As noted above, the attribute space of any state diagram consists of a single attribute corresponding to the current state of the system. It follows that the only permissible refinement of this attribute space (consistent with the definition of state diagram) is a finer grained

representation of states. That is, refinement must consist of decomposing one or more states into substates.

Consider first the case of refinements where a single state is so decomposed. It follows immediately that such a refinement must consist precisely of the "exhaustive decomposition" transformation described above: if one omits any of the possible transitions involving the newly introduced substates, one excludes behaviors which constitute refinements of the original behaviors under this decomposition, and by definition of refinement the refined state diagram must include all such behaviors.

In the most general case, a refinement may involve decomposition of several states. Clearly such a refinement can always be obtained by exhaustive decomposition of the individual states, that is, by a sequence of exhaustive decompositions. Therefore the exhaustive decomposition transformation is sufficient to generate all possible refinements of a state diagram.

Now observe that the total aggregation transformation is the inverse of exhaustive decomposition. It then follows, by arguments analogous to those just given, that total aggregation is sufficient to generate all possible abstractions of a state diagram.

So far we have demonstrated that exhaustive decomposition suffices to generate all refinements of a given state diagram and total aggregation suffices to generate all abstractions. It remains to show that these transformations suffice to relate state diagrams represented under any two frames of reference (even those not related directly by a direct chain of refinements or a direct chain of abstractions).

Consider a state diagram described under two frames of reference. As observed above, these frames of reference are defined entirely by the set of states involved in each. Let the states for the first frame of reference be $S_A = \{A_1, A_2, ..., A_m\}$ and the corresponding state diagram be

denoted by SD_A. Let the states for the second frame of reference be $S_B = \{B_1, B_2, ..., B_n\}$ and the state diagram be denoted by SD_B.

We can assume without loss of generality that there is some S such that $S = \cup A_i = \cup B_i$ so that the A_i and B_i are alternative partitions of S.[15] Then define $C_{ij} = A_i \cap B_j$.

Claim: $S_C = \{C_{ij} \mid i=1..m, j=1..n\}$ is a refinement of S_A and S_B.

Proof of claim: for all $A_i \in S_A$, $A_i \subseteq S = \cup B_j$, hence $A_i = A_i \cap S = A_i \cap \cup B_j = \bigcup_{j=1}^{j=n} (A_i \cap B_j) = \bigcup_{j=1}^{j=n} C_{ij}$. Then $C_{ij} \subseteq A_i$ for j = 1 to n. As a result, the mapping $M(C_{ij}) = A_i$ preserves state and is surjective and not bijective (assuming n > 1). Thus by definition S_C is a refinement of S_A and by analogous reasoning S_C is a refinement of S_B.

Then we can apply a set of exhaustive decompositions to SD_A to obtain the refinement SD_C (the state diagram refined under the frame of reference S_C). Then since S_C is a refinement of S_B, SD_B is an abstraction of SD_C and there is a sequence of total aggregations that when applied to SD_C results in SD_B. Then combining these results there is a sequence of exhaustive decompositions and total aggregations that when applied to SD_A results in SD_B. Since the choice of S_A and S_B was arbitrary we conclude that the set of exhaustive decompositions and total aggregations together form a complete set of refining/abstracting transformations for state diagrams.

---

[15] To see that this assumption is warranted, let $S = \cup A_i \cup \cup B_i$ so $\cup A_i \subseteq S$ and $\cup B_i \subseteq S$. If $\cup\{A_i\} \neq S$ then define an additional state $A_{m+1} = \cup(S - \cup A_i)$ and similarly for B.

Appendix E. State Diagrams: Specializing Transformations

Proposition: The following constitutes a locally complete set of specializing transformations for state diagrams:

Delete an individual event. This removes a possible transition between events and thus the new diagram is specialized to exclude all behaviors which involve such a transition.

Delete a state and its associated events. The new diagram is specialized to exclude all behaviors which involve the deleted state.

Delete an initial state marker. This transformation is subject to the condition that at least one initial state marker remains. The new diagram is specialized to exclude all behaviors which begin with the affected state.

Delete a final state marker. This transformation is subject to the condition that at least one final state marker remains. The new diagram is specialized to exclude all behaviors which end with the affected state.

Proof: For any frame of reference and any processes p_0 and p_1 described under that frame of reference, if p_1 is a specialization of p_0 then every sequence permitted in the maximal execution set of p_0 must be permitted in the maximal execution set of p_1 as well. Then all initial states of p_1 must also be initial states of p_0, and similarly for final states. Furthermore, any state or event in p_1 must be a state or event in p_0 as well, otherwise p_1 will permit a sequence involving a state or transition which cannot appear in a sequence of p_0. Thus p_0 includes all elements of p_1, and one can obtain p_1 by deleting some set of events, states, initial state markers, and final state markers. Since p_0 is itself finite, there can be only a finite number of such deletions. Thus p_1 can be obtained from p_0 by applying a finite number of transformations from the given set.

Appendix F. Formal Definition of Dataflow Diagram and Its Attribute Space

Unlike state diagrams, where a single state in an execution sequence captures the entire state of the system at that point in the sequence, a single flow or process in a dataflow diagram does not capture the state of the dataflow, which depends on the state of multiple flows and processes. In a sense the issue here is the parallelism supported by the dataflow diagram representation, where several component processes may execute simultaneously.

As it turns out, this parallelism can be captured by a single state attribute, but that attribute must take into account whether each process or flow in a dataflow diagram is currently active or inactive. A process is said to be active when it is executing (i.e. transforming inputs into outputs) and inactive otherwise. A flow is said to be active when it is available to the downstream process as an input. When a process is active it has access to those flows which are simultaneously active, and only those flows.

More formally, we define a dataflow diagram as the tuple <P,F,T,R,I,O>, where:

P is a finite set of component processes.

F is a finite set of component flows.

T is a finite set of component terminators.

R is a finite set of component stores.

P, F, T, and R must be disjoint.

I: $F \rightarrow (P \cup T \cup R)$, a function defined so that I(f) is the component which consumes flow f.

O: $F \rightarrow (P \cup T \cup R)$, a function defined so that O(f) is the component which produces flow f.

We require that all flows are either inputs to or outputs from a process. That is,

For all flows f, $I(f) \in (T \cup R) \rightarrow O(f) \in P$ and $O(f) \in (T \cup R) \rightarrow I(f) \in P$.

To define a suitable attribute space, let S denote the power set $2^{P \cup F}$, that is, the set of all subsets of $P \cup F$. Then each $s \in S$ corresponds to a possible state of the dataflow diagram in which the flows and processes in s (which, recall, is a subset of $P \cup F$) are active and all other flows and processes are not active. This list of active processes and flows is clearly an attribute of the dataflow diagram and hence S is an attribute space.

Having formalized the dataflow diagram and its attribute space, we are now in a position to formally define the maximal execution set of a dataflow diagram. First, however, we must introduce additional notation concerning dataflow diagram behavior:

Recall that a dataflow diagram behavior b is a sequence of states in S. Such a sequence can be denoted by the n-tuple $<s_1,s_2,\ldots,s_3>$. Then we define $\varphi(b,i)$ = the ith item in the n-tuple associated with b.

Then for any dataflow diagram D = <P,F,T,R,I,O>, the maximal execution set of D, henceforth denoted MES(D) is defined (consistent with our informal discussion above) as:

{b | b is a sequence of states and the two conditions defined below hold for b}

where the two conditions are:

1. We have informally asserted above that each input flow or output flow to a process which appears in b must be associated with at least one instance of that process in b. In order to formalize this condition we must elaborate it further: any flow f active in b must be active in a consecutive subsequence S_f of b such that if f is produced by a process p_1, then p_1 is active in the first state of S_f, and if f is consumed by a process p_2, then p_2 is active in the last state of S_f.[16] Formally, this condition becomes: for all s in b and for all flows $f \varepsilon F$, $f \varepsilon s \rightarrow (\exists$ m,n,p integers $0<m\leq n\leq p$) such that $\varphi(b,n) = s$ and $f \varepsilon \varphi(b,i)$ for $m \leq I$ p and $O(f) \varepsilon P \rightarrow O(f) \varepsilon \varphi(b,m)$ and $I(f) \varepsilon P \rightarrow I(f) \varepsilon \varphi(b,p)$.
2. We asserted informally above that each process in the sequence must have at least one associated input flow and one associated output flow. In formalizing this statement we restate it in slightly different form: whenever a process is active in a state s in b, then at least one input flow and output flow must also be active. Stated formally this becomes: for all $p \varepsilon$ P and for all $s \varepsilon$ b, $p \varepsilon s \rightarrow (\exists f_1,f_2 \varepsilon F)\ f_1 \varepsilon s \wedge f_2 \varepsilon s \wedge I(f_1) = O(f_2) = p$.

Henceforth we will refer to these two conditions as the *MES conditions*.

Appendix G. Refining/Abstracting Transformations for Dataflow Diagrams

We proceed by first formally defining exhaustive process decomposition and then proving that it is a refinement. Let D = <P,F,T,R,I,O> and D` = <P`,F`,T`,R`,I`,O`> be two dataflow

[16] Note that in the event f is produced (or consumed) by a terminator or store, the corresponding condition does not apply since terminators and stores are not directly included in the attribute space.

diagrams. Then we define exhaustive process decomposition as a binary relation R_P on dataflow diagrams with $R_P(D,D`)$ if and only if:

1. Replace the process to be decomposed with its subprocesses.

   $P` = (P - P^-) \cup P^+$

   where P^- is the set containing the single process component to be decomposed

   and P^+ is the set containing the subprocesses in the decomposition.

   Since P^+ must contain a generic process and at least one specific subprocess we have:

   $|P^-| = 1$ and $|P^+| > 1$.

2. Remove all flows involving the decomposed process and add all possible flows to, from, and among the subprocesses.

   $F` = F - (F_I^- \cup F_O^-) \cup F_I^+ \cup F_O^+ \cup F_+^+$

   where:

   F, F_I^+, F_O^+, and F_+^+ are all disjoint

   $F_I^- = I^{-1}[P^-]$

   $F_O^- = O^{-1}[P^-]$

   $|F_I^+| = |F_I^-|\,|P^+|$ (one new input flow per subprocess and original input)

   $|F_O^+| = |F_O^-|\,|P^+|$ (one new output flow per subprocess and original output)

   $|F_+^+| = |P^+|\,(|P^+| - 1)$ (all possible flows among subprocesses)

   and we require that none of the new flows share both input and output (i.e. no duplicate flows): $f_1, f_2 \in (F` - F) \wedge I`(f_1) = I`(f_2) \wedge O`(f_1) = O`(f_2) \rightarrow f_1 = f_2$

3. $I`: F` \rightarrow P^+$ is a function with the following properties:

   $f \in F - F_I^- - F_O^- \rightarrow I`(f) = I(f)$ (remaining original flows have same consumer)

   $I`[F_+^+] = P^+$ (internal flows have subprocesses as consumers)

   $I`[F_I^+] = P^+$ (new input flows have subprocesses as consumers)

   $I`[F_O^+] = I[F_O^-]$ (new output flows have same consumers as originals)

4. $O`: F` \rightarrow P`$ is a function with the following properties:

   $f \in F - F_I^- - F_O^- \rightarrow O`(f) = O(f)$ (remaining original flows have same producer)

   $O`[F_+^+] = P^+$ (internal flows have subprocesses as producers)

$O`[F_O^+] = P^+$ (new output flows have subprocesses as producers)

$O`[F_I^+] = O[F_I^-]$ (new input flows have same producers as originals)

5. S` is defined as the powerset $2^{F` \cup P`}$, corresponding to an attribute space for D` as described above.

Having defined exhaustive process decomposition, it remains to proved that it is a refinement:

Claim: $R_P(D,D`) \rightarrow D`$ is a refinement of D.

Proof: recall that to prove that one process representation is a refinement of another, we must prove the following three assertions:

Assertion #1. S` is a refinement of S (one attribute space is a refinement of the other).

Assertion #2. For every behavior b` ε MES(D`) there is a behavior b ε MES(D) such that b` is a refinement of b.

Assertion #3. For every behavior b ε MES(D) and every behavior b` in S`, if b` is a refinement of b, then b` ε MES(D`).

Proof of Assertion #1: By definition of refinement, it suffices to show that there is a map M: S` $\rightarrow$ S such that M is surjective, non-injective and M(s`) and s describe the same state of the world. Intuitively such an M must map the subprocesses and flows in D` to the original process and flows from which they were decomposed. More formally, we first define maps Φ and θ which take the flows in F_I^+ and F_O^+ to the original flows from which they were derived:

Φ: $F_I^+ \rightarrow F_I^-$ and $\Phi(f) = O^{-1}[O`(f)] \wedge I^{-1}[P^+]$

θ: $F_O^+ \rightarrow F_O^-$ and $\theta(f) = I^{-1}[I`(f)] \wedge I^{-1}[P^+]$

Φ and θ yields the set of all original flows with the same producer and consumer as a given f. Typically this would be a single flow (assuming no "duplicate" flows in the original dataflow diagram).

Define Γ: $(P^+ \cup F_+^+ \cup \varnothing) \rightarrow P^- \cup \varnothing$ with $\Gamma(x) = \begin{cases} \varnothing, & x = \varnothing \\ P^-, & x \in (P^+ \cup F_+^+ \end{cases}$

We will use Φ, θ, and Γ to map active flows and processes in the decomposition to active flows and processes in the original dataflow diagram and this will be the underlying basis for the map M. Recall that a state in attribute space of a dataflow diagram is a set of flows and processes which are active. Then what M needs to do is to take such a set for D` and by adding

and deleting flows and processes, convert it to the corresponding set in D. We define M as follows:

$M(s`) = s` - F_I^+ \cup \Phi[s` \cap F_I^+] - F_O^+ \cup \theta[s` \cap F_O^+] - (P^+ \cup F_+^+) \cup \Gamma(s` \cap (P^+ \cup F_+^+))$

Note that the domain of M is contained in S, since we create M(s`) by removing all elements of S - S` (subtracting out the elements of F_I^+, F_O^+, F_+^+, and P^+). Further all elements added to s` are from S (since the domains of Φ, θ, and Γ are all subsets of S). It remains to show that M is surjective and not injective.

To show that M is surjective, for any $s_0 \in S$, let

$s_1 = s_0 - P^- \cup \Gamma^{-1}[P^- \cap s_0] - F_I^- \cup \Phi^{-1}[s_0 \cap F_I^-] - F_O^- \cup \theta^{-1}[s_0 \cap F_O^-]$

Since we have subtracted out all elements of S – S` and added only elements from S`, $s_1 \in S`$. Thus we can apply M to s_1. We have carefully constructed s_1 so that $M(s_1) = s_0$, which we now prove. Note that once we have established this fact we have shown that M is surjective, since the choice of s_0 was arbitrary.

By definition of M above, we have:

$M(s_1) = s_1 - F_I^+ \cup \Phi[s_1 \cap F_I^+] - F_O^+ \cup \theta[s_1 \cap F_O^+] - (P^+ \cup F_+^+) \cup \Gamma(s_1 \cap (P^+ \cup F_+^+))$

We now evaluate several terms of $M(s_1)$ as follows:

$\Phi[s_1 \cap F_I^+] = \Phi[\Phi^{-1}[s_0 \cap F_I^-] \cap F_I^+]$, since all other terms in s_1 are disjoint with respect to F_I^+

$= \Phi[\Phi^{-1}[s_0 \cap F_I^-]]$, since $\Phi^{-1}[s_0 \cap F_I^-] \subseteq F_I^+$

$= s_0 \cap F_I^-$

By similar arguments, we have:

$\theta[s_1 \cap F_O^+] = s_0 \cap F_O^-$, and

$\Gamma(s_1 \cap (P^+ \cup F_+^+)) = P^- \cap s_0$

Substituting these results and expanding the remaining s_1 term, we have

$M(s_1) = s_0 - P^- \cup \Gamma^{-1}[P^- \cap s_0] - F_I^- \cup \Phi^{-1}[s_0 \cap F_I^-] - F_O^- \cup \theta^{-1}[s_0 \cap F_O^-] - F_I^+ \cup (s_0 \cap F_I^-) -$

$F_O^+ \cup (s_0 \cap F_O^-) - (P^+ \cup F_+^+) \cup (P^- \cap s_0)$

Noting that disjoint terms in this expression can be freely rearranged, we can reorder the terms as follows:

$M(s_1) = s_0 - P^- \cup (P^- \cap s_0) - F_I^- \cup (s_0 \cap F_I^-) - F_O^- \cup (s_0 \cap F_O^-) \cup \Gamma^{-1}[P^- \cap s_0] \cup \Phi^{-1}[s_0 \cap$

$F_I^-] \cup \theta^{-1}[s_0 \cap F_O^-] - F_I^+ - F_O^+ - (P^+ \cup F_+^+)$. Now since $s_0 - P^- \cup (P^- \cap s_0) = s_0$, we have:

$M(s_1) = s_0 - F_I^- \cup (s_0 \cap F_I^-) - F_O^- \cup (s_0 \cap F_O^-) \cup \Gamma^{-1}[P^- \cap s_0] \cup \Phi^{-1}[s_0 \cap F_I^-] \cup \theta^{-1}[s_0 \cap$

$F_O^-] - F_I^+ - F_O^+ - (P^+ \cup F_+^+)$. Simplifying further, we have $s_0 - F_I^- \cup (s_0 \cap F_I^-) = s_0$, yielding:

$M(s_1) = s_0 - F_O^- \cup (s_0 \cap F_O^-) \cup \Gamma^{-1}[P^- \cap s_0] \cup \Phi^{-1}[s_0 \cap F_I^-] \cup \theta^{-1}[s_0 \cap F_O^-] - F_I^+ - F_O^+ -$

$(P^+ \cup F_+^+)$. Substituting $s_0 - F_O^- \cup (s_0 \cap F_O^-) = s_0$, we obtain:

$M(s_1) = s_0 \cup \Gamma^{-1}[P^- \cap s_0] \cup \Phi^{-1}[s_0 \cap F_I^-] \cup \theta^{-1}[s_0 \cap F_O^-] - F_I^+ - F_O^+ - (P^+ \cup F_+^+)$.

Since s_0 is disjoint with respect to all the other terms, we can regroup the terms so that:

$M(s_1) = s_0 \cup \left(\Gamma^{-1}[P^- \cap s_0] \cup \Phi^{-1}[s_0 \cap F_I^-] \cup \theta^{-1}[s_0 \cap F_O^-] - F_I^+ - F_O^+ - (P^+ \cup F_+^+)\right)$.

Given disjoint terms we can further rearrange to obtain:

$M(s_1) = s_0 \cup \left(\Gamma^{-1}[P^- \cap s_0] - (P^+ \cup F_+^+) \cup \Phi^{-1}[s_0 \cap F_I^-] - F_I^+ \cup \theta^{-1}[s_0 \cap F_O^-] - F_O^+\right)$.

Now given the domain of Γ, Φ, and θ, we have:

$\Gamma^{-1}[P^- \cap s_0] \subseteq (P^+ \cup F_+^+)$

$\Phi^{-1}[s_0 \cap F_I^-] \subseteq F_I^+$, and

$\theta^{-1}[s_0 \cap F_O^-] \subseteq F_O^+$. Hence:

$\Gamma^{-1}[P^- \cap s_0] - (P^+ \cup F_+^+) = \varnothing$

$\Phi^{-1}[s_0 \cap F_I^-] - F_I^+ = \varnothing$

$\theta^{-1}[s_0 \cap F_O^-] - F_O^+ = \varnothing$. Substituting into M(s1), we obtain:

$M(s_1) = s_0$. Thus M is surjective.

M cannot be bijective since M maps a finite domain S` onto a finite range S and |S`| > |S|. To see this, note that $|S| = 2^{|P \cup F|} = 2^{(|P|+|F|)}$ and $|S`| = 2^{|P` \cup F`|} = 2^{(|P`|+|F`|)}$.

Now from the definition of R_P above, we have $|P`| = |(P - P^-) \cup P^+|$, but since $P^-$ and $P^+$ are disjoint and $P^- \subseteq P$, this simplifies to $|P`| = |P| - |P^-| + |P^+|$. Recall that $|P^-| = 1$ and $|P^+| > 1$. Hence $|P^-| < |P^+|$ from which it follows that |P`| > |P|.

Furthermore, again from the definition of R_P, we have $|F`| = |F - (F_I^- \cup F_O^-) \cup F_I^+ \cup F_O^+ \cup F_+^+|$ $= |F| - |F_I^-| - |F_O^-| + |F_I^+| + |F_O^+| + |F_+^+|$. Since $|P^+| > 1$ and $|F_I^+| = |F_I^-| |P^+|$, we have $|F_I^+| > |F_I^-|$ and similarly we have $|F_O^+| > |F_O^-|$, hence it follows that |F`| > |F|.

Now since |P`| > |P| and |F`| > |F|, we have $|S`| = 2^{(|P`|+|F`|)} > 2^{(|P|+|F|)} = |S|$ and thus M is not injective.

Finally, it follows directly from the definition of M that for any s` ε S`, M(s`) differs from s` only in that any active subprocesses and their associated flows are removed from s` and replaced with the corresponding process and flows in S and thus M(s`) and s` describe the same state of the world.

Thus S` is a refinement of S.

Proof of Assertion #2: Recall that we must show that b` ε MES(D`) → ∃ b ε MES(D) such that b` is a refinement of b. Let M[b`] denote the sequence obtained by applying M to each element of b` and consolidating any repeated elements in the resulting sequence. It clearly follows that b` is a refinement of M[b`] and it only remains to show that for each b` ε MES(D`), M[b`] ε MES(D). By definition of maximal execution set above, it suffices to show that M[b`] satisfies the two MES conditions which together define the relationship between active flows and active processes in the maximal execution set. That M[b`] satisfies both these conditions follows immediately from the property of refinement which insures that M(s) and s describe the same state of the world in different frames of reference. Since each process or flow in D` has a corresponding process or flow in D (albeit the mapping is many-to-one), and the input and output

relations are preserved under the refinement associated with M, then both MES conditions must be preserved by M as well.

Proof of Assertion #3: Recall that we must show that for every behavior b ε MES(D) and every behavior b` in S`, if b` is a refinement of b, then b` ε MES(D`). It suffices to show that b ε MES(D) $\wedge$ M[b`] = b $\rightarrow$ b` satisfies the MES conditions. Since as we noted in the proof of assertion #2, the MES conditions are preserved by M, this must be so, for if one of the MES conditions failed to hold for b`, it would also fail to hold for M[b`] which would contradict the assumption that b ε MES(D). Hence b` must satisfy both MES conditions.

Having proven all three assertions, the overall claim is proved: exhaustive process decomposition is a refinement.