Extended Aggregation Relationships for Process Specification and Enactment in Active Databases

Nauman Chaudhry, James Moyne, and Elke A. Rundensteiner
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122
email: {chaudhry, moyne, rundenst}@eecs.umich.edu

Abstract: Process specification in a variety of domains, such as experiment modeling, work-flow modeling, and process-flows in semiconductor manufacturing, is typically characterized by recursive specification in terms of sequences and alternatives. A variety of models have been proposed for the specification of such processes. In particular, object-oriented techniques have been used for achieving various desirable features (e.g., reusability, maintainability, etc.) in the process specification and active databases have been suggested as possible platforms for process enactment. However, on the one hand object-oriented models for process specification lack an important feature of object-orientation, namely the ability to organize processes as classes with inheritance support, and on the other hand various problems, such as lack of methodological support for active rule definition, analysis, and maintenance, stand in the way of successfully employing active database technology for process enactment. To take better advantage of both object-oriented techniques and active database technology, we present a comprehensive framework for process specification and enactment, which provides an integrated solution utilizing ideas from both these domains. This is achieved by developing PSOM (Process Specification Object Model) which is an object-oriented model with explicit aggregation constructs and extended sub-typing relationships for these new aggregation constructs. We show the use of PSOM for defining processes using the aggregation constructs and arranging these processes into class hierarchies based on the formal types of the processes. In addition, we establish guidelines for defining active rules for process enactment on PSOM process specifications. We also prove that the rule definition guidelines lead to modularized rule sets which simplify the analysis of termination behavior of active rules defined in a PSOM process database.

Keywords: Active database, object modeling, process specification, rule definition, process control.

1 Introduction

Process specification in a variety of domains, such as experiment modeling, work-flow modeling, and process-flows in semiconductor manufacturing, is typically characterized by recursive specification in terms of sequences and alternatives. Given the importance of these application domains, a variety of process specification models have been proposed in recent years. Most of these models are based on Petri nets, state charts, weighted and colored graphs, or enhancements of these basic techniques [Beck94], [Hsu96], [Leymann94], [Ellis93]. All of these techniques allow explicit representation of the process sequence structure. This explicit representation can then be directly mapped to an appropriate database schema for persistent storage of the process instances.

Object-oriented techniques and active databases have been used as implementation vehicles for a number of these proposals, with process specifications being mapped to object-oriented databases (OODBs) and active rules, encoding triggering conditions, used to carry out the enactment of the process. However, on the one hand object-oriented models for process specification lack an important feature of OODB data models, namely the ability to organize processes in class hierarchies based on sub-typing relationships between the formal types of these processes. On the other hand various problems, such as lack of methodological support for active rule definition, analysis, and maintenance, stand in the way of successfully employing active database technology for process enactment. To take better advantage of both object-oriented and active database technology, we present a comprehensive data model and framework for process specification and enactment, which provides an integrated solution utilizing ideas from both these domains. This is achieved by developing the notion of process aggregation relationships to facilitate process specification and active rule definition for process enactment.

We examine the main requirements for specifying processes and use these requirements to develop a formal process specification data model called Process Specification Object Model (PSOM). PSOM contains new type constructors which can be used to specify a process in terms of an extended (has-a) aggregation relationship between itself and sequences and alternatives of component processes. We then define sub-typing relationships for PSOM thus providing the capability of organizing processes defined using these type constructors in class hierarchies, and hence enhancing reusability of process specifications. The utility of this feature, which is an essential requirement for reuse of processes, has been recognized in the literature [Hsu96], [Buβler94]. But, to the best of our knowledge, no work has been reported as of now on developing notions of sub-typing and class inheritance for processes.

Some of the key difficulties in developing applications using active databases are the lack of methodological support for active rule definition, and the difficulty in analyzing the behavior of active rule sets [Simon95]. In this paper, we propose solutions to these different problems in the context of process enactment by exploiting particular characteristics of PSOM. We present design guidelines for defining active rules for enactment of processes specified using PSOM. We then prove that the PSOM rule definition framework leads to the definition of modularized rule sets whose termination behavior is easy to analyze. The research presented in this paper thus provides solutions for both process specification as well as process enactment.

The rest of the paper is structured as follows. In Section 2, we motivate the extensions for process specification and execution by discussing the requirements for process modeling and enactment. In Section 3, we present an extended object data model for process specification. In Section 4, the need for defining active rules for process execution and control is discussed, design guidelines for defining such rules are presented and the utility of these techniques in leading to the definition of modular and tractable rules is illustrated. We discuss the features of the proposed model and contrast it with related work in Section 5, while Section 6 contains some conclusions and directions for future work.

2 Requirements for Process Specification and Enactment

As mentioned before, process specification and enactment for a variety of domains, e.g., experiment modeling, workflow modeling, process-flows in semiconductor manufacturing, shares certain features. However, to motivate our work and define its scope, in this section we address process specification and enactment in the context of a particular domain, namely semiconductor manufacturing. In Section 5 (related work), we discuss the commonalities and differences between the requirements for this domain and various other domains.

2.1 Process Specification

Semiconductor manufacturing processes to fabricate a wafer are specified as sequences and alternatives of other processes and/or steps which need to be executed in order to fabricate that wafer [Durbeck93]. The sub-processes in turn may consist of other processes and steps, whereas steps are considered to be non-decomposable. This thus leads to a nested specification of processes with a "parent-process" at one level decomposable into a sequence of "child-processes" or "child-steps" at another level and so on. The generic term "child activity" will be used when we don't want to make a distinction between a child-step and a child-process. We also term a parent-process to be at a lower level (of nesting), and a child-activity to be at a higher-level (of nesting). The entire hierarchical structure, consisting of all the processes and steps from the lowest level of nesting to the highest, is called a process-flow. The process at the root of a process-flow corresponds to the class of wafers which may be fabricated by executing valid instances of this process-flow.

To develop the relevant concepts for process specification, we now present a simplified example of a process-flow in semiconductor manufacturing (omitting attributes to keep the example simple). Consider the process C-MOS given in [Figure 1]. This process is to be executed to fabricate C-MOS transistors. C-MOS consists of the child-process Well-formation and the child-step Metallization. Instances of Well-formation consist of a sequence of instances of the process class Wafer-cleaning, the step class Deposition, the step class Oxidation and the process class Photolitho. Since Wafer-cleaning and Photolitho are processes, each of these is further composed of child-activities. Example 1 implies that we want to be able to *specify processes as sequences of other child-activities* thus leading to nested definitions over a number of levels. C-MOS is at the lowest-level of nesting, level 0, Well-formation and Metallization are at level 1 and so on.

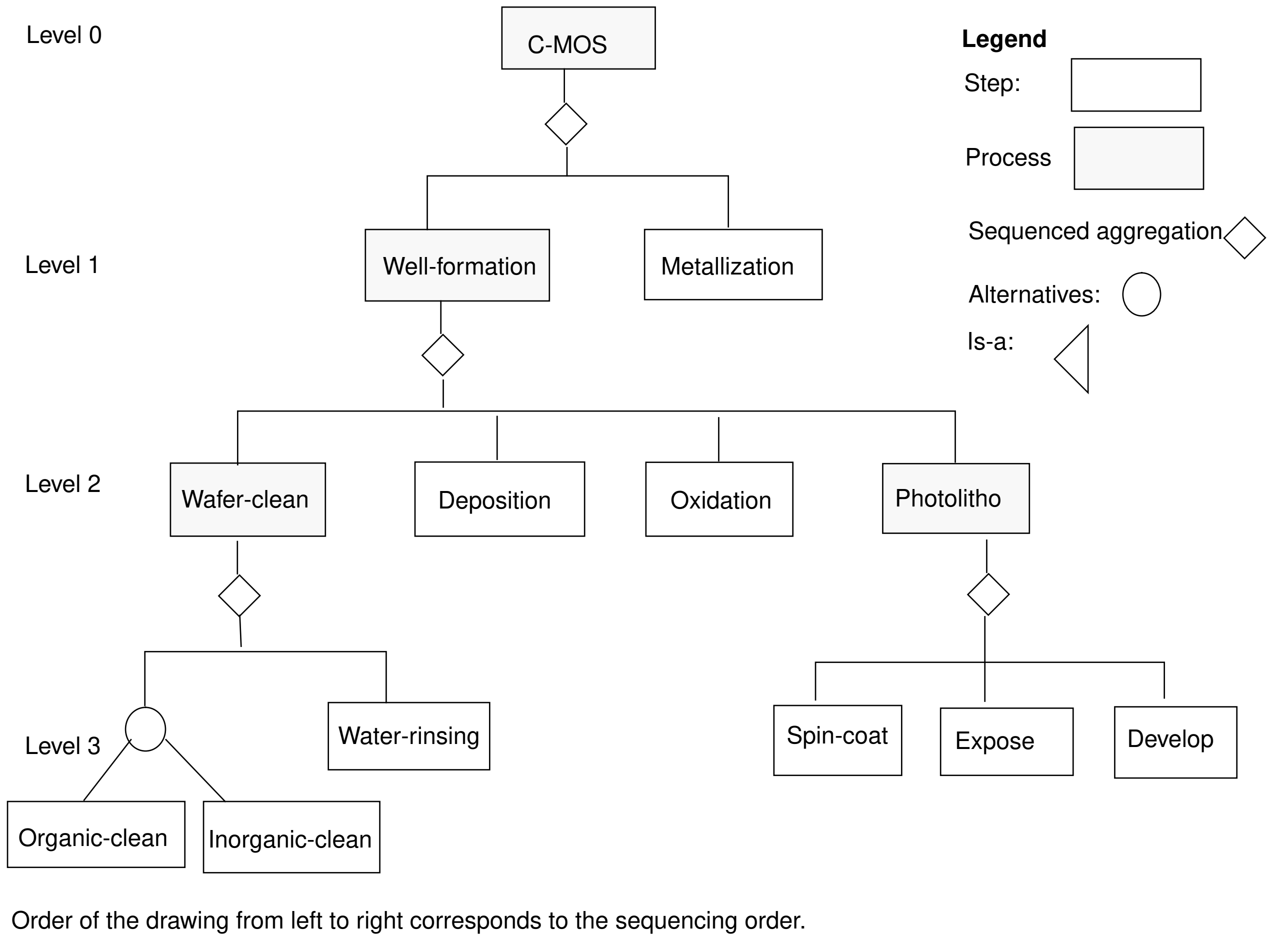

FIGURE 1. The C-MOS Process and its Descendants.

The relationship between the class of a parent-process and the classes of its child-activities is not a class-subclass relationship. For example, the Well-formation class does not generalize the Photolitho class and therefore is not a super-class of Photolitho. Rather, the Well-formation class is defined as being composed of a sequence of other (child-activity) classes. Therefore the semantics of the *relationship between a process and its child-activities are those of an aggregation relationship*, i.e., *has-a* relationship, rather than those of an *is-a* relationship. In fact, we say that a parent process is defined via a *sequenced aggregation* of its child-activities whereas conventionally has-a relationships have the semantics of being unordered [Kim89], [Liu92].

In Figure 1, note that the Wafer-cleaning process can be executed in two ways, either as a sequence of the steps Organic-clean and Water-rinsing or the sequence of steps Inorganic-clean and Water-rinsing. To handle such cases, we need to allow *specification of processes in terms of alternative sequences of child-activities.* Instances of Wafer-cleaning are then allowed to be valid instances of any of the specified alternative sequences.

Now consider that a process engineer decides to add a new class of Photolitho processes, called Photolitho-1 to the schema described above. Photolitho-1 differs from Photolitho in that Photolitho-1 has an additional child-step Post-bake after the sequence of all the child-steps of Photolitho [Figure 2]. Since the Photolitho-1 class is a refinement of the Photolitho class, this new class should be definable as a sub-class of the Photolitho class so as to profit from the advantages provided by the concept of inheritance. In OODB models for a class B to be a sub-class of another class A, the type associated with B should be a sub-type of the type associated with A [Abiteboul95]. Since the processes are defined in terms of the sequenced aggregation relationship, this requires that *the process specification model should include the notion of sub-typing for sequenced aggregation relationships.* Thus even though individual processes are specified using aggregation (i.e., has-a) relationships between a parent process and its child-activities, there is also the need to arrange processes in super-class/sub-class (i.e., is-a) relationships that make full use of the inheritance mechanisms of the object-oriented

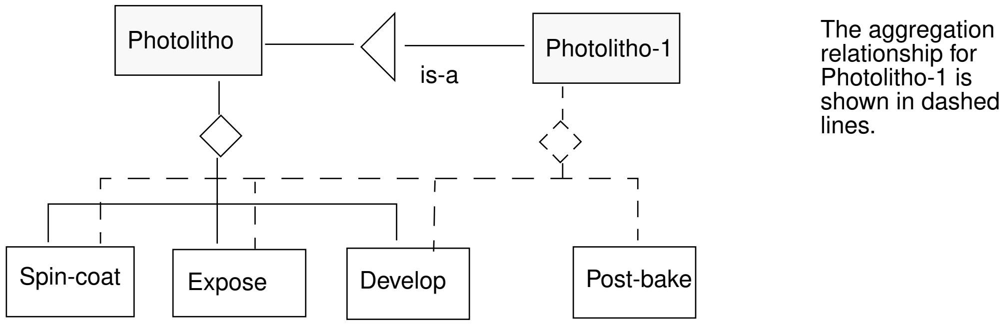

FIGURE 2. The Photolitho and Photolitho1 Processes.

2.2 Process Execution & Control

When a wafer is to be fabricated, an instance of the corresponding process is selected and the resultant process-flow is executed. The execution of the process-flow will result in the execution of the child-activities at level 1, and in the case of the child-activities that are processes, the execution in turn of the child-activities of these child-processes, and so on. The fabrication of a wafer thus results in the execution of a number of processes and steps at different levels of nesting in the process-flow. Thus *process execution has the notion of a state*, which specifies the processes and steps that have been executed at each level, those that are currently under execution, and those that are going to be executed in the future.

However, as the execution of a process-flow instance proceeds, frequently the need to control the wafer fabrication requires changes to the original processing sequence specified in the process-flow. This is caused by the fact that processes often drift with time due to a variety of factors. Performance may also change drastically after maintenance operations. Hence as there is an ongoing necessity to ensure optimal processing in the face of all these variable factors, there is a subsequent need for carrying out control over the process execution where, depending upon the state of the wafer due to prior processing, the downstream processing is modified. Examples of such changes are re-work (repeating a part of the processing sequence already carried out on the wafer to correct errors in the processing), and feed-forward control (modifying the processes yet to be carried out on the wafer to correct errors in the processing) [Larrabee91]. Thus we need the *ability to specify control actions over the process specification*. These control actions can modify processing by, for example, requiring re-execution of a part of the process-flow or by substituting parts of the original process-flow with valid instances of any alternative process sequences specified in the schema.

Active databases, with their capability to define and automatically execute rules, provide a natural implementation vehicle for process execution. Default rules can cause the execution of the different processes and steps in the original process-flow, while control knowledge can be encoded in terms of control rules. These control rules can trigger appropriate control actions during process execution whenever the need arises. As an example, consider a control rule defined to carry out re-work on the process-flow described in Section 2.1. An error occurs in carrying out the Spin-coat step of the Photolitho process. To correct this error, there is a need to re-execute the Deposition and Oxidation steps, and then carry out Expose and Develop steps of Photolitho. This control action may be encoded via the following rule:

Rule 1: ON step Spin-coat of process Photolitho of process Well-formation completes execution,

IF desired resist-thickness - actual resist thickness > α%,

DO repeat the Deposition and Oxidation steps with targets modified by α%, and then carry out Expose and Develop steps of Photolitho.

It has been noted that one of the hindrances in the acceptance of active database technology is the difficulty in defining and maintaining rule sets whose execution is predictable [Widom96]. Therefore, to present active database technology as an attractive vehicle for the implementation of process execution, the *rule programmer should be guided towards defining rules that are easy to analyze and whose behavior is predictable*.

Having discussed the requirements for process specification and enactment, in the next two section we first present an extended object model for process specification, and then describe the use of this proposed model for the definition of easy-to-analyze active rules over the process specification.

3 Process Specification Object Model (PSOM)

In this section, we present PSOM, an extended object data model that allows specification of process aggregations in terms of sequences and alternatives of processes and steps. PSOM also includes definition of extended sub-typing relationships for these process aggregations and we show the use of these features for process specification.

3.1 Base Model

We use the object-oriented data model presented in [Abiteboul95] as the base model for defining PSOM. This data model includes values and objects. A value has a type. This type is recursively definable from atomic types and the set and tuple type constructors. Object identity is one of the atomic types. An object has an identity and a state which is a value. An object belongs to a class. A value (and thus the corresponding object) can refer to other objects via their identities. Below, we give a brief description of this model, restricting ourselves to the parts of the model that we will use later. For a more detailed discussion see [Abiteboul95].

Assume the existence of a number of *atomic types* and their pair-wise disjoint domains: **integer**, **string**, **bool**, **float**. The set **dom** of *atomic values* is the union of these domains. The elements of **dom** are called *constants*. Also assume an infinite set **obj** = $\{o_1, o_2, \ldots\}$ of *object identifiers* (OIDs), a set **class** = $\{c_1, c_2, \ldots\}$ of *class names*, and a set **att** = $\{A_1, A_2, \ldots\}$ of *attribute names*. A special constant *nil* represents the undefined value. All objects in a class have complex values of the same type. The type corresponding to each class is specified by the OODB schema.

Definition 1: *Types* are defined with respect to a given set C of class names. The family of types over C is defined so that:

a) **integer**, **string**, **bool**, and **float** are types;

b) the class names in C are types;

c) if τ is a type, then $\{\tau\}$ is a (set) type;

d) if $\tau_1, \ldots, \tau_n$ are types and $A_1, \ldots, A_n \in$ **att** are distinct attribute names, then $[A_1: \tau_1, \ldots, A_n: \tau_n]$ is a (tuple) type.

The set of types over C together with the special class name **any** are denoted **types**(C). The virtual class **any** serves as the unique root of the is-a hierarchy. We associate with each class $c \in$ **class**, a type $\sigma(c)$, which dictates the type of objects in this class.

The class hierarchy has three components: 1) a set of classes, 2) the types associated with these classes, and 3) a specification of the is-a relationships between the classes.

Definition 2: A *class hierarchy* is a triple $(C, \sigma, <)$, where C is a finite set of class names, σ a mapping from C to **types**(C), and $<$ a partial order on C.

Definition 3: A *subtyping relationship* on **types**(C) is the smallest partial order $\leq$ over **types**(C) satisfying the following conditions:

a) if $c_1 < c_2$, then $c_1 \leq c_2$;

b) if $\tau_i \leq \tau_i'$ for each $i \in [1, n]$ and $n \leq m$, then

$$[A_1: \tau_1, \ldots, A_n: \tau_n, \ldots, A_m: \tau_m] \leq [A_1: \tau_1', \ldots, A_n: \tau_n'];$$

c) if $\tau \leq \tau'$, then $\{\tau\} \leq \{\tau'\}$; and

d) for each τ, $\tau \leq$ **any** (i.e., **any** is the top of the hierarchy).

A class hierarchy $(C, \sigma, <)$ is *well-formed* if for each pair c_1, c_2 of classes, $c_1 < c_2$ implies $\sigma(c_1) \leq \sigma(c_2)$.

Definition 4: A *schema* contains a well-formed class hierarchy $(C, \sigma, <)$ where σ is a mapping from C to $\mathbf{types}(C)$ of tuple type only.[1]

3.2 Extending the Object Model for Process Aggregation Relationship

We now extend the above formal model so as to explicitly model the process aggregation relationship and define sub-typing for this relationship. For examples in Section 3.2 and Section 3.3, we use the schema described in Section 2.1 (Figure 1 and Figure 2).

3.2.1 Adding New Type Constructors: SEQ and ALT

We add two new type constructors SEQ and ALT. The SEQ constructor is used to define sequences. Informally, given types $\tau_1, \ldots, \tau_n$, the SEQ type constructor defines a new type, whose values are sequences of values belonging to types $\tau_1, \ldots, \tau_n$ respectively. Formally the interpretation of type SEQ is given by:

$dom(\mathrm{SEQ}(\tau_1, \ldots, \tau_n)) = \{\mathrm{SEQ}(v_1, \ldots, v_n) \mid v_i \in dom(\tau_i), i = 1, \ldots, n\}$, and

$dom(\mathrm{SEQ}(\tau_1)) = dom(\tau_1)$.

The ALT type constructor is introduced to define alternatives. Informally, given types $\tau_1, \ldots, \tau_n$, the ALT type constructor defines a new type, whose values are the values belonging to any of the types $\tau_1, \ldots, \tau_n$. Formally the interpretation of type ALT is given by:

$dom(\mathrm{ALT}(\tau_1, \ldots, \tau_n)) = \{v_i \mid v_i \in dom(\tau_i), \text{ for some } i \in [1, n]\}$; (i.e., ALT has exclusive-or semantics).

3.2.2 The Attribute p-spec

The SEQ and ALT constructors can be used to define the child-activities of a process in terms of a sequenced aggregation of alternatives and sequences of other process and/or steps. We introduce an attribute **p-spec** (short for process specification) as a special attribute for modeling the child-activities of a process.

Example: The type of the p-spec attribute for the Photolitho class is: SEQ(Spin-coat, Expose, Develop).

By composing types defined by SEQ and ALT we can define any allowable alternative process/step sequences when specifying the child-activities of a process.

Example: The type of p-spec for the Wafer-cleaning process class can be given be as: SEQ(ALT(Organic-clean, Inorganic-clean), Water-rinsing).

3.2.3 Extended Family of Types

Types created with the proposed SEQ and ALT type constructors need to be integrated with the existing types of the object model. Our goal is to add the new type constructors SEQ and ALT so that types defined with these constructors can be meaningfully composed with each other, with the original types, and used recursively to define new process aggregations.

- To recursively build process aggregations we allow free composition of types constructed by using ALT and SEQ.
- The child-activities of a process can only be processes and/or steps, therefore we use the ALT and SEQ type constructors over only class names corresponding to process and step classes.

Types are defined with respect to a given set C of class names. To permit recursive definition and composition only among valid types, we introduce sim-types (simple types) and pf-types (process-flow types). Sim-types correspond to the types in the base model (Definition 1), whereas pf-types are introduced to specify the process aggregation relationships.

1. We ignore the presence of methods in the data model for simplicity. Note that the extensions we propose to the data model do not effect the different issues in defining methods, such as covariance, inambiguity of method inheritance, etc.

Similarly a class is termed to be either a sim-class (if the class does not model a step or a process) or a pf-class (if the class models a process or a step).

Definition 5: In PSOM the family of types over C is defined so that:

a) **integer**, **string**, **bool**, **float** are sim-types;

b) the class names in C are either sim-types (if the class is a sim-class) or pf-types (if the class is a pf-class);

c) if τ is a sim-type, then $\{\tau\}$ is a set sim-type;

d) if $\tau_1, \ldots, \tau_n$ are either sim-types or pf-types and $A_1, \ldots, A_n$ distinct attribute names, then $[A_1\colon \tau_1, \ldots, A_n\colon \tau_n]$ is a tuple sim-type.

e) if $\tau_1, \ldots, \tau_n$ are pf-types, then ALT($\tau_1, \ldots, \tau_n$) is an ALT (or alternative) pf-type;

f) if $\tau_1, \ldots, \tau_n$ are pf-types, then SEQ($\tau_1, \ldots, \tau_n$) is a SEQ (or sequence) pf-type;

The set of all sim-types and pf-types over C together with the special class name **any** are denoted **types**(C).

3.2.4 Subtyping Relationships for PSOM's Family of Types

To define the notion of sub-typing for processes we extend the sub-typing relationship (Definition 3) to the sequence and alternative pf-types constructed to define process aggregations. The definition of sub-typing for all other types in Definition 5 remains the same as in Definition 3.

Definition 6: Subtyping for Sequence Types.

In traditional object-oriented models, a sub-type needs to have all the attributes of the super-type plus possibly some additional ones. There must also be a sub-type/super-type relationship between the attributes common between the sub-type and the super-type. The same concept directly carries over for sequence types. When defining process aggregations, we require a sub-type to have all the child-activities of the super-type (in the same order) plus possibly some additional ones. There must also be a sub-type/super-type relationship between the corresponding child-activities of the sub-type and the super-type. Formally[1]:

SEQ($\tau_1, \ldots, \tau_m$) $\leq$ SEQ($\tau_1', \ldots, \tau_n'$) only if

for each τ_i' with $i \in [1, n]$, there is a τ_j with $j \in [1, m]$, such that $\tau_j \leq \tau_i'$ and

for any τ_k', τ_l' with $k < l$ and $k, l \in [1, n]$ there exist τ_o, τ_p with $o < p$ and $o, p \in [1, m]$ such that $\tau_o \leq \tau_k'$ and $\tau_p \leq \tau_l'$.

Example: The type of the p-spec for the Photolitho process class is given by: SEQ(Spin-coat, Expose, Develop). Now Photolitho-1 is defined with p-spec SEQ(Spin-coat, Expose, Develop, Post-bake). Thus the super-type/sub-type relationship holds between the p-spec attribute of Photolitho and Photliho1.

Definition 7: Subtyping for Alternative Types.

Following the definition of sub-typing for union types in [Cardelli84], the alternative sub-type can be defined by using the ALT type constructor over valid sub-types of a sub-set of types of the alternative (super) type. Formally:

if $\tau_i \leq \tau_i'$ for each $i \in [1, n]$ and $n \leq m$, then ALT($\tau_1, \ldots, \tau_n$) $\leq$ ALT($\tau_1', \ldots, \tau_n', \ldots, \tau_m'$), and

if $\tau_l \leq \tau_i$ for some $i \in [1, n]$, $\tau_l \leq$ ALT($\tau_1, \ldots, \tau_n$)

Example: Given Wafer-clean with p-spec of type SEQ(ALT(Organic-clean, In-organic-clean), Water-rinsing)) and Wafer-clean-1 with p-spec of type SEQ(Organic-clean, Water-rinsing), by Definition 7 above, the type of the child-activities of Wafer-clean-1 is a valid sub-type of the child-activities of class Wafer-clean.

Definition 5 extends Definition 1 of Section 3.1 to allow new types defined using SEQ and ALT type constructors, while Definitions 6 and 7 modify Definition 3 to extend the subtyping relationship to types defined via these new types con-

1. If the type checking for determining valid subtypes defined using the SEQ type constructor appears too costly, the sub-typing definition can be restricted so that a sub-type is allowed to add activities only at the start and the end of the super-type.

```
pf-class Process
type tuple (name: string, p-spec: sequence)

sim-class Machine
type tuple (name: string, year-of-manufacture: integer)

pf-class Expose
type tuple (name: string, expose-time: double)

pf-class Spin-coat
type tuple (name: string, resist-thickness: double, use-machine: Machine)

pf-class Develop
type tuple (name: string, expose-time: double)

pf-class Photolitho inherit Process
type tuple (p-spec: sequence(Spin-coat, Expose, Develop))

pf-class Post-bake
type tuple (name, string, temperature: double)

pf-class Photolitho1 inherit Photolitho
type tuple (p-spec: sequence(Spin-coat, Expose, Develop, Post-bake))
```

FIGURE 3. Class Definitions for the Schema of Figure 2.

structors. We leave the other definitions (Definitions 2 and 4) of Section 3.1 unchanged. Thus as in Definition 4, classes are allowed to have only the sim-type tuple.

3.3 Specifying Processes Using PSOM

Using PSOM we can now define processes as having the special attribute **p-spec** to model the process aggregation relationship plus other attributes to model the traditional attributes associated with a class.

Definition 8: A process P has a tuple sim-type, given by:

$[A_1: \tau_1, \ldots, A_n : \tau_n$, p-spec: sequence$]$ where $A_1, \ldots, A_n$ are distinct attribute names, $\tau_1, \ldots, \tau_n$ are sim-types.

Since the child-activities of a process are defined in terms of a sequenced relationship between the parent-process and the child-activities, hence the **p-spec** attribute has sequence pf-type. For the schema of a process specification database one can now define a class **Process** with the attribute p-spec (plus other attributes relevant to the application domain) and use this class as the root of the process class hierarchy. As an example, in Figure 3 we give a set of possible definitions for some of the classes shown in Figure 2 (the class Machine has been added to the schema to illustrate the co-existence of sim-classes and pf-classes).

In summary, PSOM:

- specifies processes in terms of a sequenced aggregation relationship between a process and its child-activities,
- allows specification of alternatives among the child-activities,
- is fully integrated with the base object model,
- supports sub-typing relationships over the process specification, thus allowing definition of appropriate super-class/sub-class relationships over the process classes.

We can conclude then that PSOM provides all the capabilities required for process specification as characterized in Section 2.1.

4 Process Execution & Control

Having described a comprehensive model for process specification, we now consider the issues involved in process execution and control. In the previous section, while defining processes we considered only the requirements for process specification. In this section, we first discuss the use of PSOM to define process classes that can be used for defining process execution. As briefly discussed in Section 2.2, active databases with their capability to define and automatically execute rules appear to provide an attractive implementation platform for process enactment. However, a number of issues need to be resolved for successful employment of active databases for building applications. We discuss these problems in a greater detail in Section 4.2. We then describe how PSOM can be effectively used as a basis for dealing with some of these problems. We present rule design guidelines for PSOM to address the problem of lack of design methodologies for active databases and then prove that these guidelines lead to the definition of modularized rule sets with easy-to-analyze behavior.

4.1 Using PSOM for Process Enactment

While describing PSOM we only considered the requirements for process specification. The resulting definition of the root **Process** class though adequate for specifying processes, needs to be augmented for use in process enactment. As we mentioned in Section 2.2, when a wafer is being fabricated the enactment environment needs to keep track of information relating to the process execution. In a newly instantiated process-flow all the processes are yet to be executed. On being told to begin execution by its parent process, a process P in turn tells its 1st child-activity to start execution and then waits for the child-activity to finish its execution. On being informed of this by the child-activity, P starts executing its next activity. When all the child-activities have been executed, P informs its own parent that it has finished its execution. For the process at the root of the process-flow, this means that the execution of the entire process-flow is complete. By augmenting the root **Process** class, it is possible to maintain this state information within each process as we show below.

To maintain the state information within each process, a special attribute **state** is added to the root **Process** class. The attribute **state** has the domain: "not-executed", "finished-execution", and "executing child-activity *x*", where *x* is a non-zero positive integer. The **state** attribute of each process in a newly instantiated process-flow is set to "not-executed". When a process P starts execution, the **state** is modified to "executing child-activity 1". When the child-activity finishes its execution, the **state** of P is updated to "executing child-activity 2" and so on. When all the child-activities have been executed, the **state** of P is set to "finished-execution". Note that the state variable can be treated as an integer. This can be done by using a non-zero positive value *x* to indicate that the process is in the state "executing child-activity *x*", while encoding the "not-executed" and "finished-execution" states via negative integers.

The description of process execution also points to the need of modeling another attribute of processes, namely the parent-process. This is needed so that a process can inform its parent-process on finishing its execution. An attribute **parent-process** can be added to the root **Process** class to address this need. This attribute refers to the parent (if any) of a process. Since we want processes to be reusable in defining new sequences, the type of **parent-process** should not be specific to a particular parent, but should be generic to all possible parents. It should thus be of type **Process** and should not be specialized when sub-classes of the root **Process** class are defined.

With these extensions, an augmented **Process** class that can serve as a root for the processes in Figure 1 may be defined as shown in Figure 4.

```
pf-class Process
type tuple (name: string, p-spec: sequence, state:int, parent-process: Process)
```

FIGURE 4. Definition of the Augmented Process Class.

4.2 Utilizing Active Databases for Process Execution & Control

Active database systems can monitor the database state and react to predefined situations without explicit user action or application requests. The desired behavior is expressed in terms of event-condition-action (ECA) rules. The rule syntax can be described as: *ON* event *IF* condition *DO* action, with the semantics: when the *event* occurs, if the *condition* is true, then carry out the *action*.

For carrying out process enactment, default rules can be used to cause the execution of the different processes and steps as specified in a process-flow, while active rules for control can be defined to encode control knowledge so as to trigger appropriate control actions during process execution whenever the need arises.

However, as has been noted in [Simon95], a number of problems need to be resolved to fulfill the promise of active database technology. These include:

- insufficient methodological support for designing applications using active rules,
- difficulty in analyzing interaction of control rules, and
- difficulty in maintaining such applications as rules are added and deleted.

Important properties that need to be analyzed to understand the behavior of interacting rules include termination (i.e, is rule processing guaranteed to terminate?) and confluence (i.e, is the final state of the database after rule termination independent of which rule, among multiple eligible rules, is selected for execution first?).

In case of active OODBs, there are a number of other complicating factors that are not present in relational active databases [Widom96]. These include:

- when to use active rules and when to use methods, since behavior can also be implemented using methods,
- a lack of consensus on how active rules should be integrated in the object-oriented model.

Different active OODBs use different approaches for representing rules. These include defining rules as first-class objects as, for example, in HiPAC [Dayal96], defining them within class definitions as in Ode [Lieuwen96], or defining rules as separate entities using special rule languages as in SAMOS [Geppert95]. When rules are defined as part of the schema, these are typically restricted to one class and can only specify intra-object triggers. Representing rules as first-class objects can give more expressional power, including the ability to define inter-object triggers. However as has been pointed out in [Kemper94], rules spanning multiple classes do not conform to the object-oriented paradigm, since such a definition does not respect encapsulation which is a very basic object-oriented principle. For process control one may frequently need to define rules that carry out control over multiple processes at various nesting levels.

As an example, consider Rule 1 given in Section 2.2. This rule spans multiple process and step classes. The event (step Spin-coat completes execution) is at nesting level 3, as is the condition, whereas the actions cause modifications to be carried out at both level 2 and level 3. The overall context for the rule (Well-formation process) is at nesting level 1. Clearly the rule violates the principle of encapsulation.

Programming and maintaining rules that arbitrarily span many classes can be an extremely difficult task, since the rule programmer has to ensure that a newly defined rule does not cause unforeseen interaction with other rules already defined at these different classes. There is thus a need to guide the rule programmer in defining rules that can be easily analyzed, yet have sufficient expressive power. Unfortunately, the development of generic methodologies for building applications using active database systems has been hindered due to the afore-mentioned complicating factors.

In employing active OODB technology for process enactment, we feel that there are particular characteristics of our process specification model that can be used to facilitate active rule definition, analysis and maintenance. We now show the use of PSOM as a basis for defining rules for carrying out process execution and control.

4.3 PSOM Design Guidelines for Process Execution & Control Rules

To overcome the absence of design methodologies for defining active rules, we now present guidelines for rule definition for process enactment. We want to define an appropriate scope over which a rule may be defined so that the resulting definition is tractable, while being sufficiently powerful. We also want the process classes to be reusable so that different parent processes should be able to specify the same process class as a child-process. Processes can then serve as a useful abstraction of sequences that can be reused to build new sequences.

Process classes specified using PSOM provide a foundation for defining such re-usable processes. We propose the following guidelines for defining control rules on a process specified using PSOM:

Guideline 1: To limit the scope of interaction of rules, each rule has to be defined on a particular PSOM process class. The rule can only refer to properties (i.e., attributes, methods or other rules) that are directly accessible from the process class it is defined on. This means, that the ECA parts of a rule have to be restricted to properties defined on the child-activities, on the parent, or locally on the process on which the rule is defined.

Guideline 2: Since the same process class should be re-usable in defining many different parent sequences, a rule defined on a process can only refer to those properties of the parent that are common to all possible parents, but cannot refer to properties that are specific to a particular parent. Thus when using a parent property, a a rule is restricted to the properties defined on the root **Process** class.

We have used PSOM as a foundation for rule definition. Guideline 1 causes each rule to have a well-defined scope of influence. Since we remain within the scope of the process on which the rule is defined, Guideline 1 leads to the definition of process classes which respect the principle of encapsulation. The restriction, due to Guideline 2, that only generic parent properties can be used in rule definition implies that process classes can be easily re-used in defining new sequences of processes. Additionally, since PSOM provides a powerful abstraction for all the child-activities of a process, rules defined using these guidelines avoid the lack of expressive power of conventional active rules that are defined within a class. Note that the rule guidelines are generic and are not specific to a particular rule definition language. We now show some examples of defining rules using the PSOM guidelines for default execution and for process control.

4.3.1 Rules for Default Execution

Using the rule definition guidelines, it is quite easy to define rules for default execution. We first define two events for the **Process** class to accommodate the state transitions for execution. The event "**execute**" can be raised to inform a process to start its execution, while the event "**child-finished-execution**" is added so that it may be raised by a child-activity to inform the parent that it has finished its execution. With these events defined on the root **Process** class, three rules defined on the **Process** class are sufficient for default execution as shown in Figure 5. Rule 2 is used to start the execution of a process, Rule 3 for continuing the execution of the process, while Rule 4 finishes the execution of a process when all its child-activities have been executed.[1]

Rule 2:
ON: execute,
IF: state equals not-executed,
DO: set state to 1; raise event execute for the 1st child-activity.

Rule 3:
ON: child-finished-execution,
IF: the number of child-activities > state,
DO: increment state from *x* to *x*+1; raise event execute for (*x*+1st) child-activity.

Rule 4:
ON: child-finished-execution,
IF: the number of child-activities equals state,
DO raise child-finished-execution for the parent.

FIGURE 5. Rules for Default Execution.

4.3.2 Rules for Process Control

The actual definition of rules for process control depends on the particular type of control action to be taken in a given situation. For certain control actions, such as re-work, this may require modifications to processes at multiple levels. How-

1. To handle the case where the completed process is at the root of the process-flow, at the completion of the execution of a process here is a need to check if the process indeed has a parent. We however ignore this for simplicity.

ever, our rule definition guidelines restrict the scope of each rule. But, this apparent restriction does not weaken the power of the types of control actions that may be encoded using rules. A control action spanning multiple levels may be defined via a set of rules which trigger each other at neighboring levels and we claim that this technique achieves the desired result without violating the principle of encapsulation.

Example: Rule 1 (from Section 2.2) can be refined and the desired control behavior encoded via the three rules defined to conform to our guidelines as shown in Figure 6. Rule 1a detects the error in the Spin-coat step and informs the parent. At Well-formation, Rule 1b is triggered which carries out corrective actions locally by re-executing Oxidation and Deposition, and then again passes control of execution to Photolitho. At Photolitho, Rule 1c causes corrective actions by modifying the targets of its child-activities.[1]

Rule 1a. Defined at Photolitho
ON: child-finished-execution,
IF: the child-activity is of type Spin-coat and desired resist-thickness - actual resist-thickness > α%,
DO: suspend processing; note resist-thickness error of α% at Spin-coat caused suspension;
raise event child-finished-execution for the parent.

Rule 1b.Defined at Well-formation
ON: child-finished-execution,
IF: the child-activity is of type Photolitho and is suspended due to Spin-coat thickness error,
DO: execute Oxidation and Deposition;
after these steps finish execution inform Photolitho to execute.

Rule 1c. Defined at Photolitho
ON: execute,
IF: the state is suspended due to Spin-coat thickness error,
DO: execute the remaining steps with target adjusted by α%.

FIGURE 6. Refined Definition of Rule 1.

4.4 Rule Strata for Rule Termination Analysis

The guidelines presented in the previous section lead to the definition of rules which when considered individually have a clearly specified scope of influence. Even though the *direct* interaction between rules may be limited to only the rules defined within the scope of influence specified by the rule design guidelines, when analyzing a rule set we need to also look at the possible *indirect* inter-action among rules. In the absence of additional abstractions for partitioning rules into sets with well-defined properties, the requirement to consider indirect interaction between rules makes the problem of analyzing termination properties for the rules defined in a process specification database extremely hard.

Stratification of active rules has been recently proposed as a design principle for modularization of active rules for easier termination analysis [Baralis96]. Informally, stratification consists of partitioning rules into disjoint sets or strata. Rule behavior is considered in terms of local interaction within an individual stratum and the global behavior across strata. By establishing the property of termination of rule execution in individual strata, and ordering the strata into priority levels such that the termination of higher level strata is not affected by execution of rules from lower level strata, it becomes possible to establish properties of termination for the entire set of active rules. To facilitate the termination analysis of process control rules, we now apply this concept of rule stratification to refine our rule design guidelines.

For the process control rules, the relevant stratification approach is *event-based stratification*. This approach is used to partition rules into strata and arrange these strata in a (partial) ordering of priority such that a rule stratum A is said to have higher priority level than rule stratum B (or B < A), if rules in A cannot be triggered by rules in B. It is thus possible to establish an acyclic relationship between rule sets, and as long as the rule execution in each individual stratum terminates, the execution caused by any rule triggering in the system will terminate.

1. From the example rules, the reader might have noticed that there is a need for some generic access methods for the Process class. It should be relatively straight-forward to add such methods, since the types of access is quite simple.

We now define two strata of active rules at each process class, called Det (for Detection) and Cor (for Correction). The idea is that rules belonging to the stratum Det at a process P are fired when the need for control is detected, and the rules belonging to Cor are fired when a corrective action is to be taken. The detection rules at a process P can cause corrective actions to be taken by firing correction rules at P or correction rules at children of P. These rules may also notify the parent of the need for control by firing detection rules at the parent. The correction rules at P on the other hand are allowed to fire other corrective rules at P or corrective rules at children of P. Thus we refine the PSOM rule design guidelines and add the following 3 guideline for defining active rules for a process class P:

Guideline 3: Each rule defined at a process P has to be specified as belonging to either the stratum Det or the stratum Cor.

Guideline 4: Det rules at P should be defined so that they can directly trigger only:

a) Det rules at the parent of P,

b) Cor rules at P,

c) Cor rules at children of P.

Guideline 5: Cor rules at P should be defined so that they can directly trigger only:

a) other Cor rules at P,

b) Cor rules at children of P.

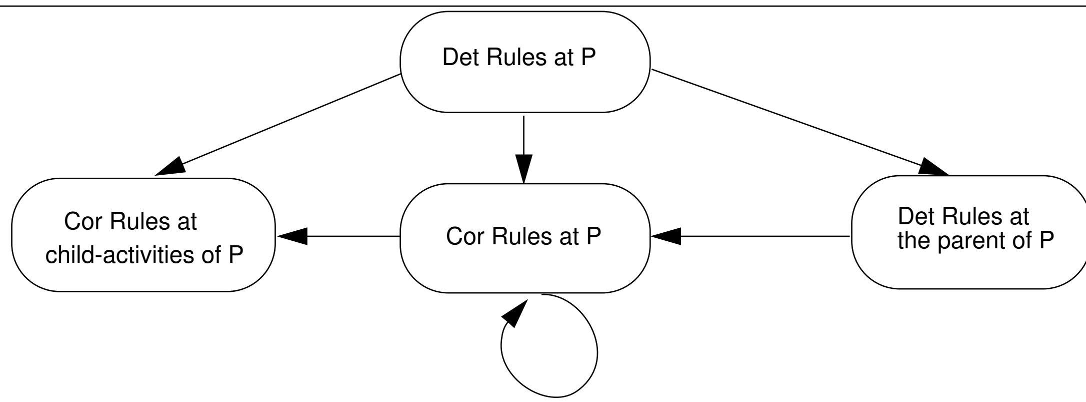

FIGURE 7. Dependency Relationships between Rule Sets which can have Direct Interaction (the arrow-head points to a set that can be triggered by the set at the other end of the line).

These restrictions imply that the Det rules at P can cause direct or indirect firing of Det rules only on those processes that are at a lower nesting level than P. Since the nesting levels in a schema are finite, the firing of Det rules will eventually terminate, as long as the firing of Det rules at each process terminates. Similarly, Cor rule firings can only be caused, either directly or indirectly, by Det or Cor rules defined at either the local level or defined at a process at a higher level. So, Cor rule firings from parent to child can again go on only along a finite number of nesting levels and as long as Cor rules at each individual process also converge, any rule execution in the active rule system will eventually converge. We thus have Proposition 1.

Proposition 1: If active rules in a process specification database expressed in PSOM respect all five PSOM rule design guidelines, and the Det and Cor rules at each individual process converge, the entire set of active rules will converge.

Proof: Given in Appendix A.

We have thus proved that the PSOM design guidelines lead to the definition of active rules whose termination behavior is easy to analyze.

5 Related Work

An extended object-oriented model for exploring semantics in aggregation relationships is presented in [Liu92], [Liu93]. Composite objects are used as a mechanism of object queries, allowing users to remain within the framework of composite objects when querying the database. Aggregation relationships are also used as a means of encapsulation by allowing the aggregate object to freely use the methods defined on constituent objects. At the type level, this results in enabling the definition of the constituent types to be re-used in the aggregate type, and this feature is termed "aggregation inheritance." We draw upon this idea by introducing the sequenced aggregation relationship and exploit it as a basis for encapsulation by using the Process type as a fundamental unit for process specification and rule definition. In addition, we have presented sub-typing relationships for sequenced aggregations. This represents a partial solution to the problem of defining sub-typing on general aggregate relationships, which is mentioned as a possible extension in [Liu92].

Process specification for a variety of others domains, in particular work-flow modeling, shares many features with the specification of process-flows in semiconductor manufacturing. Hence, the results presented in this paper will be applicable and of use to these various domains. There are, of course, additional features particular to different domains. In the case of business workflows parallel execution of processes can take place. This does not happen in semiconductor manufacturing, where the processing on the wafer can only be carried out sequentially.

Various models have been proposed for specifying work-flow processes. These include various graph-based models for process specification, including flow procedures in [Hsu96] and colored, weighted graphs in [Leymann94]. The utility of organizing processes in class hierarchies has been recognized in [Hsu96], but the issue is not addressed in either [Hsu96] or [Leymann94]. Both of these approaches allow nested definition of processes with control and data flow specification. The notion of state for process execution is also defined rigorously. The schema describes all possible variants of a process execution of a particular kind of process and triggers can be defined for picking particular variants under different conditions. Note that this differs from our approach, where the default execution is specified via a schema and control is encoded via rules. We feel that in domains like semiconductor manufacturing processes, where the possible deviations from the default execution cannot always be defined before hand and the control actions are subject to frequent change, the use of rules permits easy adaptation of the process since rules can be modified much more easily than the schema.

A fairly comprehensive object model for recursive specification of scientific experiments in terms of alternatives, sequences of and optional experiments is presented in [Chen95]. This object model has also been implemented on top of commercial relational database management systems. Though specialization and generalization of experiment models is mentioned, yet sub-typing relationships for experiment models that can be used as means of defining sub-class/super-class relationships are not discussed.

The Mentor project described in [Wodtke96] uses the formalism of state and activity charts for work-flow process specification. The emphasis is on automatically deriving an executable distributed work-flow from a formal specification and the use of state charts is of exemplary nature. The issue of arranging processes in class hierarchies is not addressed. The same remark is true of [Ellis93], in which Information Control Nets are described for use in work-flow process specification, analysis and implementation. However, the need for dynamic change and exception handling is noted, both of which we feel are provided by active rules. The use of an object-oriented database for storing work-flow specification is discussed in [Beck94]. The specification mechanism is graph-based, however process sub-class/super-class relationships are again not discussed.

Increasingly there have also been proposals for specifying processes via rules. In this approach, rules are used to define the ordering of processes, alternative processes, etc., [Kappel95]. Note that even in many of the approaches where processes are explicitly specified via graphs, rules are used to represent triggering conditions under which, e.g., one may choose between different alternative processes [Leymann94], [Hsu96]. There have also been approaches in which processes are specified via a schema, which are then processed to automatically generate active rules for process enactment according to the specified schema [Casati96].

It has been argued that the specification of processes via rules results in embedding the relationship among activities in action statements of the rules [Casati96] and it is difficult to use these rules to achieve an understanding of the progression of a process [Hsu96]. The use of specification mechanisms that explicitly represent the control flow (e.g., process schema or graphs) produces models which not only provide an intuitive understanding of the process structure, but for which notions like progression of the process can be easily defined. On the other hand, it would seem relatively simple to carry out changes when specification is done using rules, since one only needs to change the relevant rules and not the schema. Our approach, where a schema is used for specifying default execution while rules can be added for defining exceptions whenever required, provides schema specification mechanisms with the easy modification of specification via rules.

Some of the difficulties in analyzing and maintaining active rules have been mentioned in Section 4, and these criticisms would appear applicable to process specifications carried out using rules. Rule analysis is a difficult problem. Important properties of rule behavior such as termination and confluence can be very difficult or impossible to decide in the general case [Aiken95]. To help the application developer with rule programming and maintenance, modularization techniques have been proposed in [Baralis96]. For the purposes of control rule definition for semiconductor manufacturing, we have used these techniques to develop more specific guidelines and have shown that the resulting rule base can be easily analyzed to check for termination.

There has been some research carried out on the conceptual design of active OODBs [Bichler94], but comprehensive design methodologies still appear to be missing. However, there have been promising recent approaches to help the application developer with rule definition in particular domains. Rule patterns have been presented in [Kappel96] as an abstract means to capture various types of (business) control policies in a generic manner. These patterns can then be parameterized for use in specific applications. For automatically defining rules for work-flows, templated rules have also been introduced in [Casati96]. These ideas seem readily applicable to process control in general and can be used in conjunction with our rule definition guidelines to provide templates for the definition of active rules for different types of process control actions.

6 Conclusions and Future Work

In this paper, we presented a comprehensive approach for process specification and enactment. We proposed PSOM, an extended object model for process specification. PSOM includes new type constructors for defining process aggregation relationships in terms of sequences and alternatives, as well as extended sub-typing relationships for types defined using these new type constructors. This feature provides the capability of arranging process specifications in class hierarchies, thus supporting inheritance and enhancing re-use of process specifications.

For process enactment, we discussed the use of active databases and some of the problems faced in developing applications using this technology. To overcome the problem caused by the absence of generic methodologies for active rule definition, we developed guidelines for defining rules for the domain of process execution and control. These guidelines use the process aggregation relationship of PSOM as a foundation for defining rules and result in the definition of rules that remain within the scope of the process on which the rule is defined. This encapsulation feature of PSOM means that process classes with rules defined using the PSOM rule design guidelines can be easily re-used in defining various new process sequences. We then turned our attention to the problem of rule termination analysis. We refined the rule definition guidelines and proved that these refined guidelines lead to modularized rule sets for which rule termination analysis can be easily carried out. Although for descriptive purposes we have used a specific process domain, i.e., semiconductor processes, the results are applicable to other domains, in particular to work-flow processes.

We are currently in the process of implementing the proposed process specification and enactment model for developing the next version of a semiconductor process controller, first proposed in [Chaudhry95]. We developed a prototype of this controller using the Ode active DBMS. Active rules in Ode are defined as part of class definitions and are limited to a single class. This restriction limits the power of individual rules, but it does force the rule programmer to limit the scope of each rule. We have further developed this notion of well-defined scope in this paper in the form of the PSOM rule design framework. While using Ode for developing the schema for process specification, we recognized the need for developing inheritance mechanisms for our process specification model and the resulting PSOM is described in this paper.

Our work opens up a number of interesting problems for future study:

- We limit the sub-typing relationship definition to sequenced aggregations. This idea can be further explored to develop sub-typing relationships for aggregation relationships in general.

- The presented PSOM rule design guidelines deal with termination analysis of active rules sets. These can be extended so as to also provide help with confluence analysis. Perhaps this can be achieved by developing rule templates for the definition of active rules such that definition of non-confluent and non-terminating rule sets is automatically rejected.
- More use can also be made of the concept of rule stratification for defining rules for process enactment. Stratification can, for example, be used for defining rules that guarantee progression of the process execution.
- Rule inheritance in active object-oriented databases has not been fully explored. We feel that the encapsulation provided by aggregation relationships can serve as a basis for defining semantics of rule inheritance thus cleanly integrating active rule definition with the object-oriented model. This can be done by, for example, deciding what sort of changes to the sub-type could invalidate a rule defined at a super-type.

References

[Abiteboul95] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, 1995.

[Aiken95] A. Aiken, J. Hellerstein, and J. Widom, “Static Analysis Techniques for Predicting the Behavior of Active Database Rules,” ACM TODS, 20, 1, 3-41, March 1995.

[Baralis96] E. Baralis, S. Ceri, S. Paraboschi, “Modularization Techniques for Active Rule Design,” ACM TODS, 21, 1, March 1996, 1-29.

[Bichler94] P. Bichler and M. Schrefl, “Active Object-Oriented Database Design Using Active Object/Behavior Diagrams,” Proc. of the Fourth International Workshop on Research Issues in Data Engineering (RIDE-ADS ‘94), 163-171, Houston, Texas, February 1994.

[Beck94] B. Beck and S. Hartley, “Persistent Storage for a Workflow Tool Implemented in Smalltalk,” 9th Annual Conference on Object-oriented Programming, Systems, Languages and Applications, OOPSLA 94, October 94, Portland, Oregon.

[Buβler94] C. Buβler and S. Jablonski, “An Approach to Integrate Workflow Modeling and Organization Modeling in an Enterprise,” Proc. of 3rd Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, April 1994, Morgantown, West Virginia, 81-95.

[Cardelli84] L. Cardelli and P. Wegner, “On Understanding Types, Data Abstraction, and Polymorphism,” ACM Computing Surveys, 17, 4, December 1984, 471-522.

[Casati 96] F. Casati, et. al., “Deriving Active Rules for Workflow Enactment,” to appear in 7th International Conference on Database and Expert Systems Applications, DEXA 96, September 1996, Zurich, Switzerland.

[Chaudhry95]N. Chaudhry, J. Moyne, E. Rundensteiner, “A Generic Framework for Inter-Cell Control of a Semiconductor Manufacturing Facility,” 42nd National Symposium of the American Vacuum Society, Minneapolis, October 1995.

[Chen95] I. Chen and V. Markowitz, “Modeling Scientific Experiments with an Object Data Model,” Proc. 11th IEEE ICDE, 1995.

[Dayal96] U. Dayal, A. Buchmann and S. Chakravarthy, “The HiPAC Project”, in [Widom96].

[Durbeck93] D. Durbeck, et. al. “A System for Semiconductor Process Specification,” IEEE Transactions on Semiconductor Manufacturing, 6, 4, November 1993, 297-305.

[Ellis93] C. Ellis and G. Nutt, “Modeling and Enactment of Workflow Systems,” 14th International Conference on the Application and Theory of Petri Nets, Chicago, June 1993, 1-16.

[Geppert95] A. Geppert, et. al., “Architecture and Implementation of the Active Object-Oriented Database Management System SAMOS,” Technical Report 95.29, Institut fur Informatik, Universitat Zurich, Switzerland.

[Hsu96] M. Hsu and C. Kleissner, “ObjectFlow: Towards a Process Management Infrastructure,” Distributed and Parallel Databases, 4 (1996), 169-194.

[Kappel95] G. Kappel, et. al, “Workflow Management Based on Objects, Rules and Roles,” IEEE Bulletin of TC on Data Engineering, June 1995, 11-18.

[Kappel96] G. Kappel, et. al., “From Rules to Rule Patterns,” Proceedings of the Conference on Advanced Information Systems Engineering (CAiSE 96), May 1996, 99-115.

[Kemper94] A. Kemper, P. Lockemann, and H-D. Walter, “Autonomous Objects: A Natural Model for Complex Applications,” Journal of Intelligent Information Systems, 2, 133-150, 1994.

[Kim89] W. Kim, E. Bertino, and J. Garza, “Composite Objects Revisited,” Proc. of the ACM SIGMOD International Conference on the Management of Data, Portland, Oregon, May 31-June 2, 1989, 337-347.

[Larrabee91] G. Larrabee, “The Intelligent Microelectronics Factory of the Future,” Proceedings of the 1991 IEEE/SEMI International Semiconductor Manufacturing Science Symposium, 30-34.

[Leymann94] F. Leymann and W. Altenhuber, “Managing Business Processes as an Information Resource,” IBM Systems Journal, 33, 2, 1994, 326-348.

[Lieuwen96] D. Lieuwen, N. Gehani, and R. Arlien, “The Ode Active Database: Trigger Semantics and Implementation,” Proc. of the IEEE ICDE, New Orleans, Louisiana, 1996, 412-420.

[Liu92] L. Liu, “Exploring Semantics in Aggregation Hierarchies for Object-Oriented Databases,” Proc. IEEE ICDE, Phoenix, Arizona, 1992, 116-125.

[Liu93] L. Liu, “A Recursive Object Algebra Based on Aggregation Abstraction for Manipulating Complex Objects,” Data and Knowledge Engineering, 11, 1993, 21-60.

[Simon95] E. Simon and A. Kotz-Dittrich, “Promises and Realities of Active Database Systems,” Proc. of the 21st VLDB Conference, Zurich, Switzerland, 1995, 642-653.

[Widom96] J. Widom and S. Ceri, eds., Active Database Systems: Triggers and Rules for Advanced Database Processing, Morgan Kaufmann Publishers, Inc., 1996.

[Wodtke96] D. Wodtke, et. al., “The Mentor Project: Steps Towards Enterprise-Wide Workflow Management,” Proc. 12th IEEE ICDE, 556-565.

Appendix A

Proposition 1: If active rules in a process specification database expressed in PSOM respect all five PSOM rule design guidelines, and the Det and Cor rules at each individual process converge, the entire set of active rules will converge.

Proof: Let C_i and D_i be the set of Cor and Det rules defined at process *i*. Then by Guideline 5, rules in C_i cannot trigger rules in D_i, so $C_i < D_i$.

Also for all *j*, such that process *j* is at a higher level of nesting than *i*, any rule in the set D_j of Det rules at process *j* cannot be triggered either directly or indirectly by rules in either C_i (by Guideline 5) or D_i (by Guideline 4). Hence $D_i < D_j$ and $C_i < D_j$.

Similarly (by Guideline 5) for all *k*, such that process *k* is at a lower level of nesting than *i*, any rule in the set C_k of Cor rules at process *k* cannot be triggered either directly or indirectly by rules in C_i. Hence $C_i < C_k$.

However by Guideline 4, rules in the set D_i of Det rules at process i are allowed to directly trigger rules at the parent of i. Hence for all k, such that process k is at a lower level of nesting than i, rules in the set D_k of Det rules at process k may be triggered by rules in the set D_i, i.e., $D_k < D_i$.

Therefore, we get the following stratification:

Det rules at i are at a higher priority than Cor rules at i and Cor rules at all descendants and ancestors, while being lower in priority than Det rules at all descendants. Cor rules at i are at a higher priority than Cor rules at all descendants but are at a lower priority than Cor rules at all the ancestors. So for a process i in a given process-flow, we get the following event-based stratification where k is any ancestor of i, and j is any descendant of i:

$C_j < C_i < C_k < D_k < D_i < D_j$.