RULE INHERITANCE AND OVERRIDING IN ACTIVE OBJECT-ORIENTED DATABASES

Nauman Chaudhry[+], James Moyne[+], Elke Rundensteiner*
[+]Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122
***Computer Science Department**
Worcester Polytechnic Institute, Worcester, MA 01609
email: {chaudhry, moyne}@umich.edu, rundenst@cs.wpi.edu

Abstract: The concept of inheritance is among the most important features of object-oriented databases (OODBs). The different issues and trade-offs regarding inheritance have been studied for a number of years for passive OODBs. However, no general treatment of rule inheritance and overriding has been undertaken for active OODBs. Such treatment is conspicuously missing for rules that are defined over multiple classes, even though most active OODBs support the definition of such rules. In this chapter, we fill this gap by developing a formal model for rule inheritance and overriding in active OODBs. We identify important features required in an active OODB model, such as support for rule inheritance and rule overriding, and provision of the notion of syntactic compatibility. We then define a formal model for active rules by adapting the concept of multi-methods for use in defining inheritance and overriding of active rules. We extend the notion of syntactic compatibility to active rules and show how it is enforced in our formal model. The support for rule inheritance in our model is uniformly applicable to all active rules, including those rules that span across multiple classes. The work presented here thus makes the important contributions of identifying essential features missing in inheritance models of active OODB systems and of providing a formal model that incorporates these missing features.

Keywords: Active database; rule inheritance; rule overriding; formal object models.

INTRODUCTION

The concept of *inheritance* is one of the most important features of object-oriented (OO) systems. Though there are many varieties of inheritance models, inheritance fundamentally is a mechanism for sharing and incrementally modifying existing code to allow a child to inherit features (structure and behavior) from a parent (Taivalsaari, 1996). These features can then be *overridden* in the child

thus facilitating evolutionary development of complex systems. In most models, inheritance is also viewed as a facility for conceptual modeling, where a child (called a subclass) specializes its parent (called a superclass)[1]. An instance of the subclass is then said to be *substitutable* for an instance of the superclass. However, since a subclass can override the features of the superclass, the substitutability of a subclass instance for a superclass instance will only be valid if the subclass and superclass are compatible in some sense. To guarantee this conceptual correspondence, various notions of compatibility between parents and children have been established (Wegner & Zdonik, 1988, Wegner, 1990, America, 1991). By requiring that the signature of inherited properties be compatible with the superclass signature, syntactic compatibility can ensure the type-safety of existing code as new subclasses are defined. If the object model supports multiple inheritance, additional constraints are required to ensure that inheritance of features from multiple parents does not lead to ambiguity among features in a subclass.

When a passive OODB model is extended to support active rules, inheritance and overriding models must be adapted to also work with such active OODBs. While many active OODB systems support some form of rule inheritance (e.g., Ode (Lieuwen, Gehani & Arlien, 1996), SAMOS (Geppert, et. al., 1995), Sentinel (Chakravarthy, et. al., 1994)) and rule overriding is provided by (at-least) one system (TriGS (Kappel, et. al., 1994)), no general model of rule inheritance in active OODB systems has been established to-date. Such treatment is conspicuously missing for rules that are defined over multiple classes, even though most active OODBs support the definition of such rules, including, e.g., SAMOS, Sentinel, TriGS.

In this chapter, we fill this gap by developing the first formal model for rule inheritance and overriding in active OODBs. We identify important inheritance features required in an active OODB model, such as the support for rule inheritance and rule overriding, and the provision of the notion of syntactic compatibility for rule overriding. We define a model of active rules by utilizing the concept of *multi-methods* and adapting it for use in defining inheritance and overriding of active rules. We extend the notion of syntactic compatibility to active rules and show how it is enforced in our formal model. We define precise constraints on our model that ensure unambiguity in rule inheritance. This support for rule inheritance is uniformly applicable to all active rules, even if the definition of these rules spans multiple classes. After the presentation of the formal model, we discuss some pragmatic

issues regarding our model, including the issue of encapsulation in active OODBs. The work presented here makes the important contributions of identifying essential features missing in active OODB systems and then of providing a formal model that incorporates these missing features. The formal model developed is very general and can be used for extending various active OODB systems to provide support for the features of *rule inheritance* and *overriding* for both *class-specific* and *class-spanning* rules. These features can then be employed by application developers to develop active OODB applications that harness the full potential of OO technology.

The rest of the chapter is structured as follows. In the next section, we discuss the various dimensions of active rule models and identify the base active rule model that is extended later in this chapter. In the following section, we review inheritance and overriding in passive OO models, introduce an example to identify corresponding inheritance features needed in active OODBs and discuss the status of support of these features in existing active OODBs. We then outline our approach, based on the concept of multi-methods, to provide analogous support in active OODBs. After this, we present a well-known passive object data model (Abiteboul, Hull, & Vianu, 1995) that we use as a basis of our formal model for active rules. The formal OO model for active rules developed by us is then presented and is used to define inheritance, overriding, compatibility, and unambiguity of active rules in OODBs. This is followed by a discussion of certain pragmatic aspects of our model, related to encapsulation and its potential implementation to extend existing active OODBs. Conclusions are given at the end of the chapter.

RULE MODELING IN ACTIVE OODBS

The development of various active OODB systems has been carried using a variety of approaches resulting in systems that differ with each other along a number of dimensions. A pre-requisite for developing a model for active rule inheritance is an adequate underlying active rule model. To arrive at such a model, we examine the various dimensions of active rules that are important for rule inheritance and overriding. We will then identify and discuss the choices that we have selected for the active rule model used as a basis for this research.

Dimensions of Active Rule Models

A comprehensive survey of various dimensions in providing active database functionality is given in Paton, et. al. (1993). These dimensions include knowledge model (what can be said about the active rules in the system), execution model (how a set of rules is treated at run-time), and rule management model (the facilities provided by the system for managing rules). For the purposes of rule inheritance and overriding, the knowledge and rule management models are of interest. The important dimensions and possible choices for these dimensions are shown in Table 1. In the table, the symbol $\subset$ denotes that the corresponding dimension can take more than one value, whereas $\in$ denotes that the corresponding dimension can take only one of the corresponding values.

TABLE 1. Dimensions of Active Rule Models

Dimension	Possible values
Rule representation	$\subset$ {Class-internal, Class-external}
Span of rule definition	$\subset$ {Class-specific, Class-spanning}
Operations on rules	$\subset$ {Activate, Deactivate}
Transition granularity	$\in$ {Collection, Instance}
Event source	$\subset$ {Method, Abstract, Transactional, Temporal}
Event type	$\subset$ {Primitive, Composite}
Event representation	$\subset$ {Part of rule, Independent of rule}
Condition specification mechanism	$\subset$ {Query language, Programming language}
Condition representation	$\subset$ {Part of rule, Independent of rule}
Action specification mechanism	$\subset$ {Query language, Programming language}
Action representation	$\subset$ {Part of rule, Independent of rule}
Underlying data model	$\in$ {Static, Dynamic}

These dimensions are described below:

- Rule representation: A rule model in which rules are defined within class definitions is said to provide class-internal rules, whereas if rules can be defined outside a class definition (as objects or by a special rule language) then the rule definition is termed class-external.
- Span of rule definition: A rule defined for a specific class (and its subclasses) is termed a class-specific rule, while a rule defined for more than one classes (and their subclasses) is termed class-spanning rule.

- Operations on rules: Certain rule models support activate and deactivate operations on rules. If rules in an active database system need to be activated, then a rule will be applicable to an instance only if it has been explicitly activated for that instance. Otherwise, the rule will not react to event occurrences for that instance and can never be fired for it. The deactivate operation causes a (previously) activated rule to be no longer considered for firing. These two operations thus provide a means of "switching-on" or "switching-off" rules at the instance level. Alternately, these operators can also be used to disable a rule for the whole class.

- Transition granularity: Instance level granularity means that events are raised in response to operations on single data items. For rules with collection-level granularity, events are associated with operations on a collection of objects (e.g., raising an event once for all the objects obtained as the result of a query).

- Important dimensions for the event part include:

  Source of event: This dimension dictates the sources that may provide events for active rules. Possibilities include: method events, i.e., events raised as a result of the invocation of methods of database objects; abstract events, i.e., special events that can be raised without calling a method; transactional events, i.e., events raised in response to transactional operations (e.g., transaction commit, transaction abort); temporal events, i.e., events raised at particular times or time intervals.

  Type of event: Many systems provide operators to define composite events in addition to primitive events. Composite events are defined by using composition operators with primitive or other composite events. Examples of composition operators include AND, OR, Sequence, etc.

  Event representation mechanism: Event part defined as an independent entity or as part of a rule.

- For the condition, as well as the action part, important dimensions include:

  Representation mechanism: Condition (or action) part defined as an independent entity or as part of a rule.

  Specification mechanism: Condition (or action) part restricted to using only a query language or allowed to use a programming language.

- Underlying data model: The underlying object-oriented data model can have dynamic or static typing.

Dimensions of the Chosen Base Active Rule Model

Various systems use different choices for the above dimensions resulting in divergent rule models. We now (informally) state choices in the rule model that we will formalize later. These choices are also shown in Table 2.

TABLE 2. Dimensions of Our Base Active Rule Model

Dimension	Chosen values
Rule representation	{Class-external}
Span of rule definition	{Class-specific, Class-spanning}
Operations on rules	{Activate, Deactivate}
Transition granularity	{Instance}
Event source	{Method, Abstract, Transactional, Temporal}
Event type	{Primitive, Composite}
Event representation	{Independent of rule}
Condition specification mechanism	{Query language, Programming language}
Condition representation	{Independent of rule}
Action specification mechanism	{Query language, Programming language}
Action representation	{Independent of rule}
Underlying data model	{Static}

- Rule representation: Rules are defined externally to a class. This allows us to leave our base model unchanged[2].
- Span of rule definition: Rule definitions may span one or more classes, i.e., both class-specific and class-spanning rules are allowed.
- Operations on rules: Activation and deactivation operations are allowed. To be applicable to certain objects, a rule has to be explicitly activated on these objects. Additional arguments may also be passed to the rule at the time of activation (as in Ode, SAMOS, Sentinel, among others).
- Transition granularity: Transition granularity is at the instance level. This choice is consistent with most active OODB systems in which the granularity is also at the instance level (e.g., Ode, SAMOS, REACH, NAOS).
- Event source: All event sources are allowed, such as method, transaction, temporal or abstract events.

- Event composition: Events may be composed using composition operators provided by the database system (the actual set of composition operators is not important for our model).
- Event, condition and action representation: Each of the event, condition and action parts can be defined independently of a rule. This allows maximum reuse of existing definitions of these rule parts. The basic results of our model though are independent of this choice of representation.
- Condition and action specification mechanism: Each of the condition and action parts can be defined using both programming and query languages. Our results are independent of this choice of specification mechanism.
- Underlying data model: The overriding constraints are defined considering a statically typed data model. This choice is consistent with most active OODB systems, since most of these are also based on a statically-typed object model (e.g., SAMOS, Sentinel, REACH, NAOS, Ode).

BACKGROUND AND INFORMAL PROBLEM DESCRIPTION

Since inheritance and overriding of properties has already been studied in passive OODBs (Zdonik & Maier, 1990), we first review relevant research in passive OODBs. We then introduce an example to motivate the need and define the scope of the problem of rule inheritance and overriding for active OODBs. This is followed by a review of current research in rule inheritance support in active OODBs.

Inheritance and Overriding in Passive Object Models

Inheritance is a mechanism for sharing and incrementally modifying existing code where a child inherits and possibly overrides properties (structure and behavior) from a parent (Taivalsaari, 1996). In most models, inheritance is also viewed as a facility for conceptual modeling, where a child (called a subclass) specializes its parent (called superclass). An instance of the subclass is then said to be *substitutable* for an instance of the superclass. To describe the conceptual relationship between a superclass and a subclass certain compatibility rules have been defined (Wegner & Zdonik, 1988, Wegner, 1990, America, 1991). The weakest level is *cancellation*, where a subclass can redefine or even remove a superclass property. The second level is *name compatibility*, where the subclass cannot remove any properties, but may redefine them (or add new ones). The next level is *signature compatibility*, in which a subclass has full syntactic/interface compatibility with the superclass. The

strongest compatibility level is *behavioral compatibility*, where the subclass is fully behaviorally compatible with the superclass. We will refer to the first three levels as *syntactic compatibility* and focus on it in this chapter.

Even though there is no agreed upon OO data model, the various choices in defining such a model, and the impact of these choices on inheritance and subtyping in the resulting data model, are fairly well-understood (Zdonik & Maier, 1990). Decisions to have static/dynamic typing and to impose no/some constraints on the overriding of properties in the subclass are among these choices. Ideally one would like to combine static typing (to provide benefits of efficiency and type-safety) with minimum constraints on the specialization of properties in the subclass (to provide flexibility in introducing subclasses with specialized structure and/or behavior). It has been observed that if the OO system allows the definition of "mutators" (i.e., functions that can update the value of instances) on the classes, the three features of type-safety, substitutability and unrestricted specialization cannot co-exist (Zdonik & Maier, 1990). Since for almost all conceivable software systems mutators are needed, a decision has to be made about trade-offs among the desirable features. To provide unconstrained specialization, static typing may have to be given up (e.g., as done in Smalltalk, Goldberg & Robson, 1984). On the other hand, statically typed languages, with their emphasis on type-safety, impose fairly strict limits on property overriding in subclasses. Some form of syntactic compatibility, though, is provided in most OODB models. In particular, cancellation is rarely allowed and thus at least name compatibility is always provided.

To ensure that instances of subclasses are indeed substitutable in a type-safe manner, one needs to observe certain subtyping constraints that will ensure that the interface of the subclass conforms with the interface of the superclass. These constraints include (Kemper & Moerkotte, 1994):

- structure overriding in the subclass: invariance of attributes (for all attributes for which mutators are defined),
- method overriding in the subclass: co-variance (i.e., the type can be more specific) in return types, contravariance (i.e., the type can be more general) in arguments (that are not used for method binding).

It can be easily shown that if the above rules are not observed, run-time errors can occur in a statically typed system[3] (Kemper & Moerkotte, 1994).

Motivating Example

We introduce an example to motivate the need and define the scope of the problem of rule inheritance and overriding. This example is derived from semiconductor manufacturing environment (Chaudhry, Moyne & Rundensteiner, 1998). We shall use this example to determine the features an OO rule model should support for rule inheritance and overriding. For our example, we assume that:

- In the specification of the active schema, the methods whose invocation raise events are identified.
- For a given rule there may be more than one (locally defined and inherited) *rule definition* and at run-time a specific rule definition is chosen for activation. This choice will be based on some notion of "closeness" between the *formal arguments* of the chosen rule definition and the *actual arguments* of the rule activation. For a given rule name *r*, we will denote the different definitions of the rule by r_1, r_2, etc.

Consider the (simplified) schema (depicted in Figure 1) for an active OODB application for control of semiconductor manufacturing. To fabricate a product, various processing steps are executed on a wafer. These steps are carried out in a certain sequence on different pieces of equipment. Monitoring and control of these processing steps and pieces of equipment is an important activity in semiconductor manufacturing environments. Active databases can be employed to support this activity. Certain control activities can be completely automated using active rules that modify various database objects, while other activities requiring human intervention can be facilitated by active rules that alert equipment operators. The schema shown contains a `Step` class hierarchy (modeling semiconductor manufacturing steps), an `Equipment` class hierarchy (to model the equipments on which processing steps are carried out), and an `Operator` class (modeling the person who operates an equipment).

FIGURE 1

Example 1: Consider that the following two rules are defined in the active OODB.

i) For the *class-specific* rule `Pressure-Check`, a definition $\texttt{Pressure-Check}_1$ specified at the `Etch` class:

```
Pressure-Check1: Rule Pressure-Check@Etch (RIE rie)

   Event: before execute
   Condition: pressure > rie.pressure-limit
   Action: abort-execute().
```

The $\texttt{Pressure-Check}_1$ definition states that when the rule is activated on an instance of class `Etch`, an instance of the class `RIE` is expected to be passed to the rule. At the invocation of the method execute the rule condition is checked by comparing the local attribute pressure and the attribute `pressure-limit` of the given `RIE` instance to check whether the `RIE` instance is capable of providing the desired pressure. If the condition is true, then the execution of the step is aborted by the action of the rule.

ii) A definition $\texttt{Inform-Operator}_1$ of the *class-spanning* rule `Inform-Operator`:

```
Inform-Operator1: Rule Inform-Operator (Etch etch, RIE rie)

   Event: OR(after etch.execute, after rie.control-alarm)
   Condition: etch.temperature > rie.temperature-limit
   Action: rie.call-operator().
```

The rule definition $\texttt{Inform-Operator}_1$ can be activated on instances of the formal types `Etch` and `RIE`. After the method `execute` is invoked for the given instance of `Etch` or the method `control-alarm` is invoked for the given `RIE` instance, the rule condition is checked. If the condition is true, then the `Operator` of `RIE` is informed. ❒

Just as inheritance, overriding and reuse of superclass properties by subclasses is supported in the passive part of the schema, we would like to have analogous support for the active part of the schema. We will now highlight different issues to be considered for active rule inheritance by using the example code given in Figure 2 (in pseudo-C++ syntax).

FIGURE 2.

1. Rule Inheritance: Rules defined for a superclass should be inherited by its subclasses.

Example 2: A definition of the `Pressure-Check` rule has been specified at the `Etch` class and should be inherited by its subclasses. It should be possible to activate the `Pressure-Check` rule on an instance of `Chemical` (as done in `Line 09` in Figure 2). ❒

2. Overriding of Rule Parts: Just as passive properties can be overridden, overriding of rule definitions in the subclass should be allowed.

Example 3: For the class `Chemical` one could override the rule definition $\texttt{Pressure-Check}_1$ by specifying another definition $\texttt{Pressure-Check}_2$ with a modified event part:

```
Pressure-Check2: Rule Pressure-Check@Chemical (RIE rie)
   Event: OR(before execute, before set-pressure).❒
```

3. Syntactic Constraints on Overriding: In passive OODBs constraints on property overriding have been established to guarantee type safety. Similarly, there is a need to define constraints on rule overriding so that the rule definition in the subclass is syntactically compatible with the rule definition in the superclass.

Example 4: Let's assume that in the class `Physical`, the rule `Pressure-Check` were overridden so that at activation time it is passed an instance of `MXE` instead of an `RIE` instance:

```
Pressure-Check3: Rule Pressure-Check@Physical (MXE mxe)
   Event: before execute
   Condition: pressure < mxe.gas-flow-limit * 20
   ...
```

In Figure 2, the formal type of the variable `phys` is `Etch` (`Line 01`). For instances of the `Etch` class, a `RIE` instance is required when activating the rule `Pressure-Check` (definition $\texttt{Pressure-Check}_{1)}$. However, the actual type of `phys` is `Physical` (`Line 05`) and the overridden rule definition ($\texttt{Pressure-Check}_3$) for the class `Physical` expects an instance of the class `MXE`. The rule activation in `Line 11` would thus cause an invalid assignment in the absence of dynamic type checking with a `RIE` instance passed where a `MXE` instance is expected. This will result in a run-time error when in the

condition of Pressure-Check$_3$ the (non-existent) attribute gas-flow-limit is accessed for the RIE instance. ❒

4. Unambiguity in Rule Inheritance: The inheritance of active rules should be unambiguous. In the case of multiple inheritance, a subclass may inherit multiple definitions of a class-specific rule from two superclasses, resulting in ambiguity about which rule definition is applicable to the instances of the subclass. In the case of class-spanning rules, this ambiguity in inheritance of rule definitions can occur even for single inheritance.

   Example 5: Assume that rule Inform-Operator is overridden twice as follows:

```
Inform-Operator2: Rule Inform-Operator (Chemical chem, RIE rie)
   Event: OR(before chem.set-pressure, after rie.control-alarm)
   Condition: chem.temperature > rie.temperature-limit *1.2
   Action: rie.call-operator().
Inform-Operator3: Rule Inform-Operator (Etch etch, MXE mxe)
   Event: before etch.execute
   Condition: etch.temperature > mxe.temperature-limit * 1.5
   Action: mxe.call-operator().
```

The desired intent is that for instances of the class Chemical, the rule definition Inform-Operator$_2$ should be chosen, while for instances of MXE, rule definition Inform-Operator$_3$ should be applicable. Just as the run-time system in passive OODBs makes decisions about dynamic dispatch of methods, the applicability of active rules can be enforced at run-time. Thus based on the class of the *actual arguments* passed at the time of activation, one of the (three) definitions of Inform-Operator should be activated. In our example code, for the activation shown in Line 13 the definition Inform-Operator$_3$ is chosen (unambiguously) at activation time (since there is an exact match between the types of the formal and the actual arguments). The activation shown in Line 14 though results in an *ambiguous* situation:

i) for Inform-Operator$_2$ there is an exact match for argument 1, while for argument 2 the actual argument is an instance of a subclass (MXE) of the class of the formal argument (RIE).

ii) for `Inform-Operator`$_3$ again there is an exact match for one of the arguments (i.e., argument 2), while for argument 1 the actual argument (`Chemical`) is an instance of a subclass of the class of the formal argument (`Etch`). ❒

There is thus a need to constrain rule overriding so that it does not lead to ambiguity in rule applicability or alternately develop ways to disambiguate rule overridings.

Rule Inheritance in Current Active OODBs

Rule inheritance is provided by most active OODBs, although the exact mechanism for rule inheritance, and in particular rule overriding, is not widely discussed. To the best of our knowledge there has been no work on signature compatibility in active OODBs, and TriGS (Kappel, et. al., 1994) and NAOS (Collet, Coupaye & Svensen, 1994, Collet & Coupaye, 1996) are the only active OODB systems that discuss rule overriding. A key aim in TriGS is the seamless integration of active rules in an OO data model. Class-specific rules (local triggers in TriGS terminology) can be overridden. However, based on the underlying dynamically-typed object model, the rule specialization is not required to be signature compatible. Class-spanning rules (global triggers in TriGS terminology) apply to all objects that have methods corresponding to the event in the rule. Overriding of such rules though is not considered.

In NAOS (Collet, et. al., 1994, Collet & Coupaye, 1996), rules are defined as part of the schema and do not belong to a class. It is noted that providing rules as part of a class definition leads to schema update problems when modifying rules. User-defined event types are defined independently from a rule definition and belong to the schema, uniquely identified by their names. For rule overriding it is stated that the event for an overridden rule should be a sub-event type of the event of the original rule, though the meaning of subtyping for events is not presented. Similarly, there is no discussion of whether and how class-spanning rules may be overridden in NAOS.

In the SAMOS system, both class-specific and class-spanning rules are supported (Gatziu, Geppert & Dittrich, 1991, Geppert, et. al., 1995). Class-specific rules (class-internal rules in SAMOS terminology) can be encapsulated within the class and may not be visible to the user of the class. Events can be defined so that they have independent existence and can be reused in multiple rules. Class-spanning rules (class-external rules in SAMOS terminology) are identified with event

definitions rather than class definitions. Rules are inherited by the subclasses. The semantics of inheritance for class-spanning rules and rule overriding though have not been discussed. The rule model for Sentinel (Chakravarthy, et. al., 1994) is fairly similar to the SAMOS rule model. In Chimera (Ceri, et. al., 1996, Meo, Psaila & Ceri, 1996) again a distinction is made between targeted rules (with events belonging to a single class, i.e., class-specific rules in our terminology) and un-targeted rules (class-spanning rules in our terminology). Issues related to rule overriding though are not discussed. In REACH (Buchmann, et. al., 1995, Zimmermann, et. al., 1996) rules are defined independently of class definitions and at the conceptual level are considered as n-ary relationships between classes involved in the ECA parts. However, by considering rule inheritance only in relation to events, but not to rules as a unit, rule overriding is not considered (Zimmermann, et. al., 1996). In Ode, rules are defined within the definition of a class and are inherited by subclasses (Lieuwen, et. al., 1996). Overriding of rules is not supported.

A recent work in rule inheritance and overriding in active OODBs, with particular reference to the Chimera active OODB (Ceri, et. al., 1996), is described in (Bertino, Guerrini & Merlo, 1997). The focus of this work is on, what we have termed, semantic compatibility for class-specific rules. Inheritance of class-spanning rules or issues of syntactic compatibility though are not investigated.

RATIONALE OF OUR APPROACH

As discussed in the previous section, current active OODB systems have overlooked the important aspect of inheritance and overriding of class-spanning rules. We now propose that the concept of multi-methods provides a natural building block on which a model for inheritance and overriding of class-spanning (as well as class-specific) active rules can be developed. In this section, we give a brief introduction to multi-methods. We then give an outline of our approach to building a formal object model for rule inheritance and overriding.

Multi-Methods

In most OO models, a method dispatch is based on the actual type of a single object - the receiver. The actual type of the rest of the arguments in the invocation is not used in this decision of which method to invoke in response to the message. This *single-dispatching* style works well for many

kinds of messages. But for cases where the number of "interesting" arguments is more than one, single dispatching requires awkward work-arounds.

Example 6: Consider that the method `displayOn(Shape s, Device d)` displays the given `Shape s` on the given `Device d`, and that the particular implementation to be chosen depends on the actual types of both `s` and `d`. With single dispatching, the programmer has to decide which argument is more important and dispatch based on that. Then within the implementation of each of the singly-dispatched methods, dispatch based on the actual type of the other argument has to be hand-coded. ❐

Multi-methods provide a solution for such cases where single-dispatching poses a problem. In multi-methods, multiple arguments to a message can be used in method dispatch (Chambers, 1992). The programmer can define a number of methods with arguments that are more specific than the arguments of a more general method with the same name. Then depending upon the actual type of multiple arguments in an invocation, the method whose arguments most closely match those of the invocation is dispatched at run-time. However, an ambiguity may arise if there is no single method for which the match in argument is closer than all the other methods. Various techniques have been developed for carrying out (dynamic or static) type-checking of multi-methods to determine if the definition of multi-methods is ambiguous and to suggest or automatically carry out disambiguation (Agrawal, DeMichiel & Lindsay, 1991, Amiel & Dujardin, 1996). Note that singly-dispatched methods are a special case of multi-methods, i.e., multi-methods where the dispatch is based on only one argument (the receiver).

Outline of Approach

We now outline our approach of leveraging off multi-methods to develop a model for active rule inheritance and overriding. Consider three activations of rules on the schema illustrated in Figure 1 and reproduced below from the code fragment shown in Figure 2:

```
// activate rule Pressure-Check on instance chem
Line 09: chem->Pressure-Check(rie);

// activate rule Inform-Operator on the instances etch & mxe
Line 13: Inform-Operator(etch,mxe);

// activate rule Inform-Operator on the instances chem & mxe
```

```
Line 14: Inform-Operator(chem,mxe);
```

In `Line 09`, a class-specific rule is activated. This activation is very similar to a method dispatch and it is easy to see that the notions of inheritance, overriding and compatibility can be developed for such rules by extending the corresponding notion for methods. We observe however that the activations of the class-spanning rules, as shown in `Line 13` and `Line 14`, correspond in some sense to the dispatch of a multi-method. Depending upon the actual type of multiple arguments in an activation, the rule whose arguments most closely match those of the activation should be activated at run-time. Just as singly-dispatched methods are a special case of multi-methods, we propose that class-specific rules be viewed as a special case of class-spanning rules.

The activation of rule `Pressure-Check` in `Line 09` is then stated in our model as:

```
Line 09: Pressure-Check(chem, rie); // activate rule Pressure-Check on instance chem
```

with activation dynamically carried out based on the actual type of only the first argument.

We thus state the key insight - *the concept of multi-methods can be adapted to define a model for rule inheritance and overriding*. In our model, rule definitions are specified with “links” to the event, condition and action parts, each of which is specified independently. Importantly, the span of rule definition for each rule in our model is specified via a designated set of classes. Inheritance of rules is then defined based on subclass relationships between this designated set of classes and other classes in the passive schema. With the above formulation, we are then able to define the notions of syntactic compatibility and unambiguity of inheritance for active rules.

A BASE PASSIVE OBJECT MODEL

We now develop a formal model for rule inheritance in active OODBs. As our base model, we adapt the well-known object model presented in (Abiteboul, Hull & Vianu, 1995). This object model is presented in this section. In the next section, we will add appropriate constructs to this model to define active rule inheritance.

Informal Description of Base Object Model

We first informally describe the basic entities of our base object model. The formal definitions of these concepts follow in the next section.

- This data model includes values and objects (Definition 1). A value has a type. This type is recursively definable from atomic types and the set and tuple type constructors (Definition 2).
- 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.
- A schema is specified by a user via a set of classes (modeling the application), the type of each of these classes (which can be defined in terms of the base classes and other user-defined classes by using the type constructors) and a subclass relationship for these classes (Definition 3).
- The data model includes definition of sub-typing relationships over a given set of classes (Definition 4).
- The user defined sub-class relationships have to respect these sub-typing relationships (Definition 5).
- The behavioral part of the schema is defined via a set of well-formed methods (Definition 6 and Definition 7).

Formal Object Model

Values and Object Identifiers

Assume the existence of a number of *atomic types* and their pair-wise disjoint corresponding domains: **integer**, **string**, **bool**, **float**. The set **dom** of atomic values is the (disjoint) 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.

Definition 1: Given a set O of OIDs, the family of values over O is defined so that

a) *nil*, each element of **dom**, and each element of O are values over O; and

b) if $v_1, \ldots, v_k$ are values over O, and $A_1, \ldots, A_n$ distinct attribute names, the tuple $[A_1: v_1, \ldots, A_n: v_n]$ and the set $\{v_1, \ldots, v_n\}$ are values over O.

The set of all values over O is denoted by **val**(O). An object is a pair (o, v) where o is an OID and v a value. ❒

Passive Types

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 2: Passive 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**, **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 $A_1, \ldots, A_n$ distinct attribute names, then $[A_1: \tau_1, \ldots, A_n: \tau_n]$ is a (tuple) type. ❒

We associate with each class c a type $\sigma(c)$, which dictates the type of objects in this class. In particular, for each object (o, v) in a class c, v must have the structure described by $\sigma(c)$.

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

Definition 3: A *class hierarchy* is a triple (C, σ, $\leq_S$), where C is a finite set of class names, σ a mapping from C to **types**(C), and $\leq_S$ a partial order on C. ❒

Definition 4: A subtyping relationship on **types**(C) is the smallest partial order $\leq_T$ over **types**(C) satisfying the following conditions[4]:

a) if $c_1 \leq_S c_2$, then $c_1 \leq_T c_2$;

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

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

c) for each τ, $\tau \leq_T$ **any** (i.e., **any** is the top of the hierarchy). ❒

Definition 5: A class hierarchy (C, σ, $\leq_S$) is *well-formed* if for each pair c_1, c_2 of classes, $c_1 \leq_S c_2$ implies $\sigma(c_1) \leq_T \sigma(c_2)$. If $c_1 \leq_S c_2$ and $c_1 \neq c_2$, we say that $c_1 <_S c_2$. ❒

Adding Behavior

The final ingredient of the generic OODB model is *methods*. A method has three components:

a) a name,

b) a signature,

c) an implementation (or body).

The names and signatures of methods are specified at the schema level in our OODB model. We assume the existence of an infinite set **meth** of method names. Let (C, σ, $\leq_S$) be a class hierarchy.

Definition 6: For method name m, a signature of m is an expression of the form m: $c \times \tau_1 \times \ldots \times \tau_{n-1} \rightarrow \tau_n$, where c is a class name in C and each τ_i is a type over C. The class c is termed the *receiver* of m. A set M of methods is *well formed* if it obeys the following two rules:

- Unambiguity: If c is a subclass of c' and c'' and there is a definition of m for c' and c'', then there is a definition of m for a subclass of c' and c'' that is either c itself, or a superclass of c.
- Co/contravariance: If $m : c \times \tau_1 \times \ldots \times \tau_n \rightarrow \tau$ and $m : c' \times \tau_1' \times \ldots \times \tau_p' \rightarrow \tau'$ are two definitions, $c <_S c'$ and $n = p$, then for each i, $\tau_i' \leq_T \tau_i$ and $\tau \leq_T \tau'$ (i.e, contravariant in the formal arguments and covariant in the return type). ❒

Schemas and Databases

Definition 7: A passive schema is a 4-tuple $S = (C, \sigma, \leq_S, M)$, where

- σ is a mapping from C to **types**(C) of tuple type only;
- $(C, \sigma, \leq_S)$ is a well formed class hierarchy; and
- M is a well-formed set of method signatures for $(C, \sigma, \leq_S)$. ❐

Example 7: The set of classes C for the schema in Figure 1 is: {`Step, Etch, Chemical, Physical, Equipment, RIE, MXE, Operator`}. The definition of types of some of the classes and the specification of the subclass relationship is given below:

`Step` = [`name: string`]

`Etch: super-class Step` = [`int: temperature, int: pressure`],

`Equipment` = [`name: string, operated-by: Operator`].

The set of method signatures M is given by: {`abort-execute: Etch` $\rightarrow$ `int, execute: Etch` $\rightarrow$ `RIE, execute: Physical` $\rightarrow$ `MXE, set-pressure: Chemical` $\rightarrow$ `int, call-operator: Equipment` $\rightarrow$ `int, control-alarm: RIE` $\rightarrow$ `int, inform: Operator` $\rightarrow$ `int`}. The set is well-formed, with the two definitions of execute conforming to co/contravariance and unambiguity. Note that in the examples to follow, the return type of methods will be left unspecified to simplify the presentation. ❐

ACTIVE RULES

We now define a formal model for active rules which will be used to define syntactic compatibility for rule inheritance and overriding. Active rules are specified with reference to a passive schema and include:

a) a set of *event names* and the *types* for these events names;

b) a set of *conditions*;

c) a set of *actions*;

d) a set of *rule names* and *a set of definitions* for each rule name.

Some Useful Definitions

Class Vectors: To simplify the presentation of our model, we will use $\overline{c}^{(n)}$ to represent a vector of classes $c_1, \ldots, c_n$.

Subclass Relation among Vectors: We will also use the notation $\overline{c}^{(n)} \leq_S \overline{b}^{(n)}$ to represent that the classes in the vectors $\overline{c}^{(n)}$ and $\overline{b}^{(n)}$ and stand in the relation $\leq_S$ point-wise, i.e., $\overline{c}^{(n)} \leq_S \overline{b}^{(n)} \equiv c_i \leq_S b_i$ for $i = 1, \ldots, n$.

Specificity of Vectors: If $\overline{c}^{(n)} \leq_S \overline{b}^{(n)}$ and for at-least one $i \in [1, n]$, $c_i \neq b_i$, we say that $\overline{c}^{(n)}$ is more specific than $\overline{b}^{(n)}$, denoted by $\overline{c}^{(n)} <_S \overline{b}^{(n)}$.

Most Specific Vector: Given class vector $\overline{c}^{(n)}$ and a set $\text{Vec}^{(n)}$ of class vectors (all of the same size), $\overline{s}^{(n)} \in \text{Vec}^{(n)}$ is said to be most specific w.r.t. $\overline{c}^{(n)}$, if

$\overline{c}^{(n)} \leq_S \overline{s}^{(n)}$, and for all other $\overline{b}^{(n)} \in \text{Vec}^{(n)}$ if $\overline{c}^{(n)} \leq_S \overline{b}^{(n)}$ then $\overline{s}^{(n)} <_S \overline{b}^{(n)}$

Note that for an arbitrary set of class vectors, a most specific vector may not exist.

Events

Event types are defined using the primitive events and the composition operators provided by the system. Primitive events may include method events, transaction events, temporal events and abstract events. Of these, method events are related to the passive schema. Method events are identified by the schema designer by designating that an event be raised before/after the invocation of a particular method in the passive schema. Different composition operators are defined by various systems. Examples of composition operators include conjunction, disjunction, and sequence.

Assume the existence of a set E_m of method events of the form either *before m* or *after m* where $m \in M$; a set E_{temp} of temporal events; a set E_{abs} of abstract events; and a set E_{trans} of transaction events. We designate the set of primitive events by E_{prim}, i.e., $E_{prim} = E_m \cup E_{temp} \cup E_{abs} \cup E_{trans}$. We also assume a set $OP = \{op_1, op_2, \ldots, op_l\}$ of composition operators.

Definition 8: Event types are defined with respect to a given set of primitive events E_{prim} and operators belonging to the set OP, such that

i) each element of E_{prim} is an event type;

ii) $op(e_1, e_2, \ldots, e_q)$ is a (composite) event type if $e_1, e_2, \ldots, e_q$ are event types that can be legally composed using $op \in OP$.

The set of event types defined for a given set of primitive events E_{prim} and operators belonging to the set OP is denoted by **events**(E_{prim}, OP). ❒

Example 8: In the schema shown in Figure 1, four method events have been identified. Thus E_m = {`before execute: Etch, after execute: Etch, before set-pressure: Chemical, after control-alarm: RIE`}.

Using the composition operator `OR`, the composite event `OR(after execute: Etch, after control-alarm: RIE)` and the composite event `OR(after control-alarm: RIE, before set-pressure: Chemical)` have been defined in the rule definitions `Inform-Operator`$_1$ and `Inform-Operator`$_2$ respectively (in Examples 1 and 5). ❒

To simplify event definition, the return type of the primitive events will be left out in the future.

Conditions

The condition part of an active rule is a query over the database. Queries are specified by Boolean expressions in a query language and/or a programming language. The expression may range over attributes and methods of a set of classes (on which the condition is defined) and may include comparison operators, quantifiers and connectives provided by the query language. The interface of a condition will be included in the rule definition. This is modeled by a name, a signature (both of which are at the schema level) and a body (which is at the instance level). The signature of a condition contains the classes on which the condition is specified (i.e., the classes whose attributes and methods are used in the condition body) and the parameters that may be passed to the condition at evaluation time.

Definition 9: For each condition named *cond*, there is a signature of *cond* which is an expression of the form *cond*: $c_1 \times \ldots \times c_k \times \tau_1 \times \ldots \times \tau_{p-1} \rightarrow$ **bool**, where $c_1, \ldots, c_k$ are class names in C and $\tau_k \in \sigma(C)$, for $k = 1, \ldots, p - 1$. The set of condition names for a given active schema is denoted by M_C. ❒

Example 9: The body of the condition for rule definition `Pressure-Check`$_1$ (Example 1) is `pressure > rie.pressure-limit`. The condition is defined on the class `Etch` and it uses an instance of `RIE` in its evaluation. Its name and signature are given: `cond`$_1$`: Etch` $\times$ `RIE` $\rightarrow$ **bool**. The name and signature of the condition for rule definition `Inform-Operator`$_1$ (Example 1) is given by: `cond`$_2$`: Etch` $\times$ `RIE` $\rightarrow$ **bool.** ❒

Since the return type of all conditions is **bool**, we will leave it unspecified in examples in the future.

Actions

Actions in our model are methods defined in the given passive schema *S*.

Definition 10: For each action named *action*, there is a signature of *action* $\in$ *M*. The set of action names for a given active schema is denoted by the set M_A. ❒

Example 10: The signature of the action of rule definition `Pressure-Check`$_1$ (again leaving the return type unspecified) is `abort-execute: Etch`. ❒

Rules

Rule types are defined using a rule constructor with respect to a given set *R* of rule names, a given passive schema *S*, the set $E \subset$ **events**(E_{prim}, *OP*) of events, the set M_C of condition names, and the set M_A of action names. We will denote the various definitions for a rule name *r* by r_1, r_2, etc.

Definition 11: For each rule $r \in R$, a type definition (or simply definition) r_i of *r* is given by a *rule type* which has the following form:

r_i = <name: *r*, class$_1$: c_1, . . ., class$_n$: c_n, event: *e*, condition: f_c, action: f_a, par$_1$: τ_1, . . ., par$_p$: τ_p>

where $c_i \in C$, for $i = 1, \ldots, n$; $e \in E$, $f_c \in M_C$, $f_a \in M_A$, $\tau_k \in \sigma(C)$, for $k = 1, \ldots, p$, and for each method event in (the primitive or composite) *e*, the receiver c_m of the method appears in $c_1, \ldots, c_n$. ❒

Here *e*, f_a, and f_c specify the event, condition and action of the rule respectively. c_1 to c_n represent the classes to which the rule applies. An instance of each of these classes will be passed to the rule at activation time. The par$_k$ represents any other parameters passed to the rule. The restriction on the

method events, stated in Definition 11, ensures that only those method events that are raised by methods of the classes to which the rule applies can effect the rule firing.

Example 11: The type of the two rule definitions introduced in Example 1 is as follows:

`Pressure-Check`$_1$ = <`name: Pressure-Check, class`$_1$`: Etch, event: before execute: Etch, condition: cond`$_1$`: Etch` $\times$ `RIE, action: abort-execute: Etch, par`$_1$`: RIE`>.

`Inform-Operator`$_1$ = <`name: Inform-Operator, class`$_1$`: Etch, class`$_2$`: RIE, event: OR(after execute: Etch, after control-alarm: RIE), condition: cond`$_2$`: Etch` $\times$ `RIE, action: call-operator: RIE`>. ❒

Definition 12: The term **class-vector**(r_i) represents the classes to which the definition r_i applies, i.e., **class-vector**(r_i) = $\overline{c^{(n)}}$. ❒

Example 12: The class vector of the definition `Inform-Operator`$_1$ is: **class-vector**(`Inform-Operator`$_1$) = <`Etch, RIE`>, while the class vector of the definition `Pressure-Check`$_1$ of rule `Pressure-Check` is given by: **class-vector**(`Pressure-Check`$_1$) = <`Etch`>. ❒

Rule Inheritance

We first consider the case of rule inheritance in the absence of overriding. A rule *r* defined on a given class-vector is inherited by all the class vectors that are more specific than this vector.

Definition 13: A rule definition r_i, with | **class-vector**(r_i) | = l is applicable to and hence can be activated for instances of all class vectors $\overline{a^{(l)}}$ belonging to $C \times C \times \ldots \times C$ (l-times), such that $\overline{a^{(l)}} <_S$ **class-vector**(r_i). The set of all such class vectors $\overline{a^{(l)}}$ is denoted by **Inherits**(r_i). ❒

Example 13: The definition `Inform-Operator`$_1$ of rule `Inform-Operator` is defined on the class vector: <`Etch, RIE`>. This rule is inherited by the class vectors defined by: <`Etch, MXE`>, <`Chemical, RIE`>, <`Chemical, MXE`>, <`Physical, RIE`>, <`Physical, MXE`>. This means that this rule definition applies to and can be activated on instances of any of these class vectors. ❒

Rule Overriding

A rule definition can be overridden by specifying another rule definition with the same name but a different type. Some or even all of the event, condition and action attributes may be modified in this

overridden rule. The scope of the overridden rule definition is specified via a different class-vector which represents the classes for which the overridden rule definition is applicable.

Definition 14: Given rule definitions r'_j = <name: r, $class_1$: c'_1, . . ., $class_n$: c'_n, event: e', condition: f'_c, action: f'_a, par_1: τ'_1, . . ., par_p: τ'_p> and r_j = <name: r, $class_1$: c_1, . . ., $class_n$: c_n, event: e, condition: f_c, action: f_a, par_1: τ_1, . . ., par_p: τ_p>, if $\overline{c'}^{(n)} <_S \overline{c}^{(n)}$ then r'_j overrides r_j. ❒

Example 14: Consider the two definitions $\texttt{Pressure-Check}_1$ and $\texttt{Pressure-Check}_2$ of rule `Pressure-Check` (Example 1 and Example 3 respectively). These are given in our formal model as:

$\texttt{Pressure-Check}_1$ = <name: Pressure-Check, $\texttt{class}_1$: Etch, event: before execute: Etch, condition: $\texttt{cond}_1$: Etch × RIE, action: abort-execute: Etch, $\texttt{par}_1$: RIE>.

$\texttt{Pressure-Check}_2$ = <name: Pressure-Check, $\texttt{class}_1$: Chemical, event: OR(before execute: Etch, before set-pressure), condition: $\texttt{cond}_1$: Etch × RIE, action: abort-execute: Etch, $\texttt{par}_1$: RIE>.

Here $\texttt{Pressure-Check}_2$ overrides $\texttt{Pressure-Check}_1$ for the class `Chemical`. The scope of the rule definition labeled $\texttt{Pressure-Check}_2$ is specified with the class-vector <Chemical> and <Chemical> $<_S$ <Etch>. The event is overridden by a composite event, while the condition and action parts are left unchanged[5]. ❒

Ensuring Compatibility

When the definition of a rule is overridden by specifying another definition, we need to ensure compatibility of the new definition with the existing definitions of the rule. To achieve this, we require that each rule must have a *most-generic* definition and any definition that overrides the most-generic definition must be *compatible* with this most-generic definition. Similarly, if certain definitions already override a most-generic definitions, then any new definition must be mutually-compatible with all these existing definition. This notion is made precise below.

Definition 15: For each rule $r \in R$, a particular rule type definition r_i must be identified as most-generic. The corresponding class-vector is termed **generic-class-vector**(r). ❒

Example 15: For the rule `Pressure-Check`, we specify that the rule definition $\texttt{Pressure-Check}_1$ is most-generic. Thus **generic-class-vector**(`Pressure-Check`) = <`Etch`>. For the `rule Inform-Operator`, we specify that the rule definition $\texttt{Inform-Operator}_1$ is most-generic. Thus **generic-class-vector**(`Inform-Operator`) = <`Etch, RIE`>. ❒

Obviously, for overriding to make sense, the class vector of the more specific rule definition should itself be more specific than the class vector of the most-generic definition.

Definition 16: For each rule name $r \in R$, the *rule overridings* of r is a set of rule definitions (denoted by **Overriding**(r)) of the following form:

<name: r, class$_1$: c'_1, . . ., class$_n$: c'_n, event: e', condition: f'_c, action: f'_a, par$_1$: τ_1, . . ., par$_p$: τ_p>
where $c'_i \in C$, for $i = 1, \ldots, n$, $\tau'_k \in \sigma(C)$, for $k = 1, \ldots, p$; $e' \in E$, $f'_c \in M_C$, $f'_a \in M_A$ and $\overline{c'}^{(n)} \leq_S$ **generic-class-vector**(r). ❒

Example 16: The rule overridings for `Inform-Operator` are the definitions $\texttt{Inform-Operator}_1$, $\texttt{Inform-Operator}_2$ and $\texttt{Inform-Operator}_3$ with the **class-vectors** <`Etch, RIE`>, <`Chemical, RIE`> and <`Etch, MXE`> respectively. Note that the most-generic definition of $r \in$ **Overriding**(r). ❒

Name Compatibility

For passive object models, name compatibility requires that a subclass may redefine a superclass property (or add new properties), but it may not remove any superclass properties. The analogous concept in the case of rules is that if a rule r is applicable to (and hence can be activated on) instances of a class c, and $c' <_S c$ then r should also be applicable to instances of c'.

Proposition 1: Consider a rule r defined with a certain class-vector $\overline{c}^{(n)}$ that includes a class c. Then Definition 13 specifies that this rule can be activated on the class-vector $\overline{c'}^{(n)}$ in which c is replaced by any subclass of c (since $\overline{c'}^{(n)}$ is more specific to $\overline{c}^{(n)}$, and hence by Definition 13 it is included in **Inherits**(r)). Name compatibility is thus ensured by our model. ❒

Signature Compatibility

Signature compatibility for passive object models implies full compatibility of the interface of a class with the interface of its superclass. In the application code, a rule definition is chosen for activation on certain objects and the required parameters are passed to this rule definition. Thus for each rule definition, the interface consists of the formal types of the classes on which the rule definition can be activated and the types of the parameters that can be passed to the rule definition. By adapting the definition of compatibility for multi-methods (Castagna, 1995), we can see that signature compatibility requires that in all overridings of a rule *r* the class vector should become more specific (i.e., covariant) and the parameter overriding can only be made less specific (i.e., contravariant).

Definition 17**:** Two type definitions of r, r'_j = <name: r, $class_1$: c'_1, . . ., $class_n$: c'_n, event: e', condition: f'_c, action: f'_a, par_1: τ'_1, . . ., par_p: τ'_p> and r_j = <name: r, $class_1$: c_1, . . ., $class_n$: c_n, event: e, condition: f_c, action: f_a, par_1: τ_1, . . ., par_p: τ_p> are said to be *signature compatible* if $\overline{c'}^{(n)} <_S \overline{c}^{(n)}$, then also for each i, $\tau_i \leq_T \tau'_i$. The *rule overridings* of a rule r exhibit signature compatibility if each of them is mutually signature compatible. ❒

Example 17: For the rule `Pressure-Check` consider the three definitions $\texttt{Pressure-Check}_1$, $\texttt{Pressure-Check}_2$, and $\texttt{Pressure-Check}_3$ in Examples 1, 3 and 4 respectively. Here:

class-vector($\texttt{Pressure-Check}_1$) = <`Etch`> and activation of $\texttt{Pressure-Check}_1$ expects an instance of `RIE` as its parameter,

class-vector($\texttt{Pressure-Check}_2$) = <`Chemical`> and activation of $\texttt{Pressure-Check}_2$ expects an instance of `RIE` as its parameter,

class-vector($\texttt{Pressure-Check}_3$) = <`Physical`> and activation of $\texttt{Pressure-Check}_3$ expects an instance of `MXE` as its parameter.

Now class-vector($\texttt{Pressure-Check}_2$) $<_S$ class-vector($\texttt{Pressure-Check}_1$) and `RIE` $\leq_T$ `RIE`, therefore $\texttt{Pressure-Check}_1$ and $\texttt{Pressure-Check}_2$ are signature compatible. However, class-vector($\texttt{Pressure-Check}_3$) $<_S$ class-vector($\texttt{Pressure-Check}_1$), but `RIE` $\leq_T$ `MXE` does not hold true. The rule overridings of `Pressure-Check` are thus not signature compatible, since the parameter overriding is not contravariant. Note that if $\texttt{Pressure-Check}_1$ and $\texttt{Pressure-Check}_2$ were the only

definitions of rule `Pressure-Check` then the rule overridings of `Pressure-Check` would be signature compatible. ❒

Unambiguity of Rule Inheritance

Unambiguity in rule inheritance requires that for a given rule *r* and a given class vector there should be a unique definition of *r* that is applicable to this class vector. This can be enforced by requiring that a set of rule overridings be well-formed as defined below.

Definition 18: A set of rule overridings of *r* is well-formed if for each $\overline{a}^{(n)} \in$ **Inherits**(*r*), there exists a $\overline{c}^{(n)} \in$ **Overridings**(*r*) which is most-specific w.r.t. $\overline{a}^{(n)}$. ❒

Example 18: Consider the set of overridings `Inform-Operator`$_1$, `Inform-Operator`$_2$, `Inform-Operator`$_3$ specified for the rule `Inform-Operator`. Observe that a most specific class-vector <`Etch, MXE`> exists w.r.t. the class vector <`Physical, MXE`>. However, for the class vector <`Chemical, MXE`> no such class vector exists (since <`Chemical, MXE`> $<_S$ <`Etch, MXE`> and <`Chemical, MXE`> $<_S$ <`Chemical, RIE`> and neither <`Etch, MXE`> $<_S$ <`Chemical, RIE`> or <`Chemical, RIE`> $<_S$ <`Etch, MXE`>). This set of overridings thus does not meet our condition of being well-formed. If now another definition of `Inform-Operator` is specified with the class-vector <`Chemical, MXE`>, the set of overridings of `Inform-Operator` becomes well-formed. ❒

DISCUSSION

Our basic rule model, though formal, is similar to the informal model described for the SAMOS system in (Gatziu, et. al. 1991, Geppert, et. al., 1995). In SAMOS events, conditions and actions can be defined independently of rules. Both class-specific and class-spanning rules are supported and are inherited by the subclasses. As mentioned earlier, the semantics of inheritance for class-spanning rules and rule overriding though have not been discussed for SAMOS. In addition, the concepts of name and signature compatibility have not been investigated in SAMOS or any other active OODB system that we are cognizant of.

It should be noted here that it has been argued elsewhere that class-spanning rules violate the notion of encapsulation and hence do not match well with the OO paradigm (Kemper, et. al., 1994). However, the utility of class-spanning rules is obvious and these rules are provided by almost all active OODBs[6]. In fact, this provision of class-spanning rules (which is analogous to multi-methods) need not completely violate encapsulation. The key concept for encapsulation of rules is lexically restricted scope of access to a class. This can be achieved via constructs similar to the "friend" construct of C++, which gives access to private properties of a class to a function defined outside this class (Stroustrup, 1991).

For active rules specified using the model presented in the chapter, checking for ambiguous rule definition and resolving such ambiguities becomes an important task. Type checking of multi-methods can be carried out statically using polynomial-time algorithms presented in, e.g., Chambers & Leavens (1995). The same basic algorithm would be applicable for our rule model. Similarly ambiguity in rule definition can be solved in a manner similar to the corresponding solution in the area of multi-methods. Different mechanisms for the definition of most specific methods are discussed in Amiel & Dujardin (1996). These include, for example, establishing precedence based on argument order, precedence based on inheritance order, or explicitly asking the user to specify the precedence. The first two mechanisms are examples of implicit disambiguation, where there is an automatic resolution of ambiguity without requiring programmer input. Requiring programmers to solve ambiguity is called explicit disambiguation. It has been argued in Amiel & Dujardin (1996) that implicit disambiguation generally hides programmer errors that are the cause of ambiguity in the first place. The authors thus argue for explicit disambiguation. In case of rule inheritance, we feel that a similar argument suggests that rule disambiguation should also be carried out explicitly. This will ensure that the rule programmer is aware of the interaction among inherited rules and can make use of this fact when disambiguating the rules.

CONCLUSIONS

Inheritance, though a key feature of OO systems, has been surprisingly under-explored in active OODBs. This has resulted in active OODB systems lacking the important features of overriding of class-spanning rules and notions of syntactic compatibility. In this chapter, we have presented a formal model which covers the overlooked aspects of inheritance and overriding of class-spanning

rules and also lays down a formal foundation of the concept of syntactic compatibility for rule overriding. This model has been developed by adapting the concept of multi-methods and can be used in the implementation of active OODB systems to provide the presented functionality.

For future work, we note that the focus of the active rule inheritance model introduced in this chapter has been on defining the notion of syntactic compatibility for rule inheritance and overriding. For this model to be implemented in an active OODB system and hence be available for use to application developers, there is a complementary need to provide language support for rule inheritance. Extensions can be carried out to rule definition languages to provide reuse of rule specifications by supporting the definition of a more specific rule in terms of an existing rule that is being overridden. Compilers for such extended rule languages can provide type checking to ensure syntactic compatibility, thus providing complete support for the formal model defined in the chapter.

Lack of support for inheritance and overriding is not the only feature for which support for active capabilities lags the corresponding support for passive capabilities. There has been little or no research for providing support for active schema evolution. The area of schema evolution has been researched in the context of passive OODB systems (Zicari, 1992, Ra & Rundensteiner, 1995). There is a need for developing a corresponding framework to support schema evolution in active databases. Rule-base, and in particular event-base, modification has been studied in Geppert, Gatziu & Dittrich (1995). The emphasis is on carrying out the evolution of composite events while taking into account the events that have already taken place before the evolution. However, a general framework, encompassing operators for active schema modification and notions of consistency for active and passive schema is missing. We feel that a framework for schema evolution in active OODB systems can be developed using the formal model for active rules presented in this chapter.

REFERENCES

Abiteboul, S., Hull, R. & Vianu, V. (1995). *Foundations of Databases*. Reading, MA: Addison-Wesley.

Agrawal, R., DeMichiel, L. & Lindsay, B. (1991). Static Type Checking in Multi-Methods. *Proc. of 6th Annual Conference on Object-oriented Programming, Systems, Languages and Applications,* 113-128.

America, P. (1991). In M. Lenzerini, D. Nardi, & M. Simi (Eds.), *Inheritance Hierarchies in Knowledge Representation and Programming Languages.* New York: John Wiley & Sons.

Amiel, E. et. al. (1996). Type-safe Relaxing of Schema Consistency Rules for Flexible Modelling in OODBMS. *The VLDB Journal,* 5 (2), 133-150.

Amiel, E. & Dujardin, E. (1996). Supporting Explicit Disambiguation of Multi-Methods. *Proc. of 10th European Conference on Object-Oriented Programming,* 167-188.

Bertino, E., Guerrini, G. & Merlo, I. (1997). Trigger Inheritance and Overriding in Active Object Database Systems. *Proc. of the 5th International Conference on Deductive and Object-Oriented Database,* 193-210.

Buchmann, A. et. al. (1995). Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions. *Proc. of 11th IEEE International Conference on Data Engineering,* 117-128.

Castagna, G. (1995). Covariance and Contravariance: Conflict without a Cause. *ACM Transactions on Programming Languages and Systems,* 17 (3), 431-447.

Ceri, S. et. al. (1996). In J. Widom & S. Ceri (Eds.). *Active Database Systems: Triggers and Rules for Advanced Database Processing.* San Francisco, CA: Morgan Kaufmann.

Chakravarthy, S. et. al. (1994). *ECA Rule Integration into an OODBMS: Architecture and Implementation* (Report No. UF-CIS-TR-94-023). Dept. of Computer & Information Science, University of Florida, Gainesville, Florida.

Chambers, C. (1992). Object-Oriented Multi-Methods in Cecil. *Proc. of 6th European Conference on Object-Oriented Programming*, 33-56.

Chambers, C. & Leavens, G. (1995). Typechecking and Modules for Multi-Methods. *ACM Transactions on Programming Languages and Systems*, 17 (6), 805-843.

Chaudhry, N., Moyne, J. & Rundensteiner, E. (1998). Active Controller: Utilizing Active Databases for Implementing Multi-Step Control of Semiconductor Manufacturing. *IEEE Transactions on Components, Packaging and Manufacturing Technology, Part C: Manufacturing Technology*, to appear.

Collet, C., Coupaye, T. & Svensen, T. (1994). NAOS - Efficient and Modular Reactive Capabilities in an Object-Oriented Database System. *Proc. of 20th International Conference on Very Large Databases*, 132-143.

Collet, C. & Coupaye, T. (1996). Composite Events in NAOS. *Proc. of 7th International Conference on Database and Expert Systems Applications*, 244-253.

Gatziu, S., Geppert, A. & Dittrich, K. (1991). Integrating Active Concepts into an Object-Oriented Database System. *Proc. of 3rd International Workshop on Database Programming Languages*, 399-415.

Geppert, A., Gatziu, S. & Dittrich, K. (1995). *Rulebase Evolution in Active Object-Oriented Database Systems: Adapting the Past to Future Needs* (Report No. 95.13). Computer Science Department, University of Zurich, Switzerland.

Geppert, A. et. al. (1995). *Architecture and Implementation of the Active Object-Oriented Database Management System SAMOS* (Report No. 95.29). Computer Science Department, University of Zurich, Switzerland.

Goldberg, A. & Robson, D. (1984). *Smalltalk-80: the Interactive Programming Environment.* Reading, MA: Addison-Wesley.

Kappel, G. et. al. (1994). TriGS - Making a Passive Object-Oriented Database System Active. *Journal of Object-Oriented Programming,* 7 (4), 40-63.

Kemper, A. et. al. (1994). Autonomous Objects: A Natural Model for Complex Applications. *Journal of Intelligent Information Systems,* 3 (2), 133-150.

Kemper, A. & Moerkotte, G. (1994). *Object-Oriented Database Management: Applications in Engineering and Computer Science.* Englewood Cliffs, NJ: Prentice Hall.

Lieuwen, D., Gehani, N. & Arlien, R. (1996). The Ode Active Database: Trigger Semantics and Implementation. *Proc. of the 12th International Conference on Data Engineering,* 412-420.

Melton, J. (1996). SQL-3 Update. *Proc. of the 12th International Conference on Data Engineering,* 666-672.

Meo, R., Psaila, G. & Ceri, S. (1996). Composite Events in Chimera. *Proc. of 5th International Conference on Extending Database Technology,* 56-76.

Paton, N. et. al. (1993). Dimensions Of Active Behaviour. *Proc. of the 1st International Workshop on Rules in Database Systems,* 40-57.

Ra, Y-G. & Rundensteiner, E. (1995). A Transparent Object-Oriented Schema Change Approach Using View Evolution. *Proc. of 11th IEEE International Conference on Data Engineering,* 165-172.

Stroustrup, B. (1991). *The C++ Programming Language.* (2nd ed.). Reading, MA: Addison-Wesley.

Taivalsaari, A. (1996). On the Notion of Inheritance. *ACM Computing Surveys,* 28 (3), 438-479.

Wegner, P. (1990). Concepts and Paradigms of Object-oriented Programming. *ACM OOPS Messenger,* 1 (1), 7-87.

Wegner, P. & Zdonik, S. (1988). Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. *Proc. of the 2nd European Conference on Object-Oriented Programming*, 55-77.

Zdonik, S. & Maier, D. (Eds.) (1990). *Readings in Object-Oriented Databases*. San Francisco, CA: Morgan Kaufmann.

Zicari, R. (1992). In F. Bancilhon, C. Delobel & P. Kanellakis (Eds.), *Building an Object-Oriented Database System, The Story of O2*. San Francisco, CA: Morgan Kaufmann.

Zimmermann, J. et. al. (1996). Design, Implementation and Management of Rules in an Active Database System. *Proc. of 7th International Conference on Database and Expert Systems Applications*, 422-435.

AUTHOR BIOGRAPHIES

Nauman Chaudhry received his B.Sc. (E.E.) degree from The University of Engineering and Technology, Lahore, Pakistan, in 1991, and his M.S.E. degree in Computer Science & Engineering from The University of Michigan, Ann Arbor, in 1994. He is currently a Ph.D. candidate at the Software Systems Research Laboratory at The University of Michigan. His doctoral research has been focussed on extending database technology for manufacturing and control applications. His research interests include object-oriented and active databases, data models for imprecise data, work-flow and manufacturing process automation.

James Moyne received his B.S.E.E. and B.S.E. - Math, M.S.E.E. and Ph.D. degrees from The University of Michigan in 1983, 1984 and 1990 respectively. His dissertation title is *System Design for Automation in Semiconductor Manufacturing*. Since 1992 he has been employed as an Assistant Research Scientist in the Department of Electrical Engineering and Computer Science at The University of Michigan – Ann Arbor. His research interests and publication topics include discrete manufacturing (run-to-run process and inter-process) control, sensor bus device and network modeling, database design and implementation, and network modeling, protocol development and

standardization. He also authored a number of communications standards for the semiconductor industry. Additionally, he has a background in local area networks, solid-state circuits, and mathematics, and has a patent on a generic cell controller design which is currently being used in discrete process control in the semiconductor manufacturing industry.

Elke Angelika Rundensteiner is currently Associate professor of the Department of Computer Science at the Worcester Polytechnic Institute, after having been a faculty member in the Department of Electrical Engineering and Computer Science at the University of Michigan for several years. She has received a BS degree (Vordiplom) from the Johann Wolfgang Goethe University, Frankfurt, West Germany, an M.S. degree from Florida State University, and a Ph.D. degree from the University of California, Irvine. Dr. Rundensteiner has been active in the database research community since over 10 years. Her current research interests include object-oriented databases, data warehousing, database and software evolution, multi-media databases, distributed and web database applications, GIS, and information visualization. She has more than 100 publications in these areas. Her research has been funded by government agencies including NSF, ARPA, NASA, CRA, DOT; and by industry including IBM, AT&T, Informix and GE. She is regularly on Program Committees of the key conferences in the database field such as IEEE ICDE, ACM SIGMOD, VLDB, as well as others. Dr. Rundensteiner has received numerous honors and awards, including a Fulbright Scholarship, an NSF Young Investigator Award, and an Intel Young Investigator Engineering Award, and an IBM Partnership Award.

---

[1] Since the meaning of and relation between certain object-oriented terms (e.g., subclass, subtyping) varies in different object models, note that in our chosen object model a subclass relationship between two classes implies the existence of subtype relationship between the types of these classes. This appears to be the common approach in OODBs, although certain OO programming languages (e.g., Cecil in Chambers (1992)) decouple subclass relationship (mechanism for code inheritance) and subtype relationship (mechanism for interface inheritance).

[2] In type-theoretic terms, we have an overloading-based object model as opposed to a record-based object model (Castagna, 1995). However, as noted in Castagna (1995), ignoring implementation issues, these two ways of grouping methods are essentially equivalent at the formal level, and hence this choice does not restrict the formal power of our model.

[3] To handle cases where these rules appear too strict, techniques have been developed which allow some type unsafe statements with conditions and exception handling code around them (Amiel, et. al., 1996). Alternately the user may be encouraged to follow the constraints without the system enforcing them completely (e.g., C++ (Stroustrup, 1991)).

[4] To ensure type safety, the legal subtyping relationships defined here are different from the one presented in (Abiteboul, et. al., 1994), and correspond to the GOM model defined in (Kemper & Moerkotte, 1994).

[5] With a suitable rule definition language one should be able to define a more specific rule in terms of the existing rule that is being overridden. Similarly, it is possible to give short names to the events and use these names in the rule definition. This issue though is orthogonal to our formal model.

[6] It may be noted here that SQL3 includes multi-methods (Melton, 1996). A requirement that rules do not span classes thus seems undue if the semantics of the application require such functionality.