

Object methods for data organisation

P. Herth and Jim E. Newton

Cadence Design Systems

(Dated: June 10, 2005)

In this paper, the application of object oriented methods to data organization is described. Applying the methods known from code structuring to data management creates a set of powerful tools to manage complex data sets in a structured way. To demonstrate the concept the described methods have been applied to an example in ICDesign, the EPM (Extensible PCell Methodology) system. The resulting EPM system showed unparalleled ease and flexibility of customization of PCells through class definitions of the configuration data.

I. INTRODUCTION

Object orientation is a very common technique to structure program logic. By building a hierarchical class structure, common code parts need only to be programmed once and can be shared by all classes which inherit them. This approach has been successfully used to manage the complexities of huge codebases. This paper discusses the idea of applying these methodologies onto data organisation and the resulting consequences.

II. OBJECT ORIENTATION FOR CODE

For software projects containing large sets of different data types, the observation can be made, that usually these data types are not totally independent of each other, but rather certain relationships can be found. A schoolbook example of this is the management of graphical shapes. The different shapes, like rectangles and circles, share an amount of common properties. The classical object orientation is a means to express these relations in the code structure as well as try to use this knowledge to create smaller and more specific code.

The building block of the object systems are classes. Classes define a set of slots for objects, in which they hold their data, in analogy of structured data types. Furthermore they allow the definition of methods upon the class type to create class-specific code. So code and data slots are aggregated together by the concept of the class. While the concepts of slots or instance variables is a common trait of object systems, the ways of specifying the methods vary between the different programming languages. Languages like Java and C++[1] have an object model, where methods are members of the classes they are specified on. They live in the namespace within the class and only dynamically dispatch on the class type they are defined on. However, statically the methods can be overloaded upon their parameter types.

In the CLOS Common Lisp object system[2], methods do not belong to certain classes, but have a function like notation with the dispatch expressed by the types of their parameters. They reside in the namespace of the package they are defined in rather than in a namespace of a class. The CLOS system allows methods to dynamically

dispatch on any number of parameters.

The key concept of object orientation is inheritance. Inheritance means that instead of creating each class from scratch and having to duplicate the overlapping code or data parts, one can create new classes as a derivative of existing classes in the system. That means, one specifies only the difference of the new class to the existing classes it is derived from. This does not only save coding effort but also increases the robustness of the system. Each feature is specified only once, and thus it is ensured, that any change or bugfix is propagated into all classes inheriting this feature.

The simplest model for inheritance is single inheritance. With single inheritance, a class can specify one parent class which it inherits from. The slot and method definitions then only add to or overwrite those of the parent class. The object systems of languages like Java and Smalltalk are of the single inheritance model. Single inheritance models have the disadvantage of not allowing mixin classes for adding certain featuresets to several class hierarchies.

Other object systems like C++, Python and CLOS allow multiple inheritance. Here, when creating a new class, more than one ancestor class can be specified, which makes for a more complex inheritance concept. A very common operation is to define utility classes, which have no use of their own, but are intended to be mixed into other classes to provide them with certain features. Multiple inheritance allows a mixin class to be inherited into any number of class hierarchies. Without limiting the general case, from here on inheriting from two ancestor classes is assumed when discussing multiple inheritance, though any number is allowed. The simple case is, where the ancestor classes have no overlap. That means, that all their slots differ in names and have no methods in common. In this case, the newly created class is just an union of its ancestors slots and methods.

However, when there is an overlap between the ancestor classes, then strategies are required to resolve this overlap. First step in this process is to compute the class precedence list, which orders the list of all ancestor classes. The simplest approach to resolving overlaps, is take the most specific class from the precedence list, and take its definition of a slot or method as the one to select, overwriting all possible other ones. CLOS tries to go a bit further than that, by merging all information about a

slot that gets defined in more than one ancestor classes. That is, the accessors and initfunction and arguments are combined, only when more than one ancestor class gives for example an initarg, the one of the ancestor class with the highest precedence overwrites the others. Beyond this, no automatic merging can be done, since only the most specific method can be computed, but not the code of the methods be changed.

III. OBJECT ORIENTATION FOR DATA

Complex data sets can show the same hierarchical internal structure like code sets. So it is quite logical to approach their representation by the same methods, which are now a standard for code organisation. That means, to create new data sets by inheriting from previous ones and either adding new entries to it or modify existing entries. This alone gives the data set representation a very nice structure. However, when inspecting the conflict resolving strategies known from the multiple inheritance model in the code domain, completely new opportunities appear. The conflict resolution for code is determined by the fact, that you cannot programmatically change code, for example merge it. The conflict resolution can only select one of the methods as the applicable one. With data however, the situation completely changes, because data can be modified by functions. So what was a means to resolve conflicts only in code inheritance, now becomes an important feature of the system.

CLOS makes an attempt for more sophisticated conflict resolution by allowing to define method combinations so that the return values of methods can be merged or combined[3]. It cannot of course perform operations on the code within the methods to be combined.

Setting up an object system for data comes quite natural in Lisp due to its `defmacro` facility[4]. So the data definitions can blend in seamlessly with the rest of the code and it is possible to have code parts executed as part of the data definition.

The basic operator of a data object system looks like the `defclass` of CLOS, a class is created by listing its name, classes inherited from and then a list of slots. The difference here is, that the slots do not represent variables, but for named data fields. Hence, each slot is directly initialized with a value. Furthermore, it has to be specified, what kind of data is stored in the slot. To resolve conflicts resulting from inheriting the same slot from different sources with different values, inheritance rules have to be given. In the following, it is shown how these functions are applied in an industrial IC Design application called EPM.

IV. EXTENSIBLE PCELL METHODOLOGY

A layout used for semiconductor production consists of small scale geometrical shapes on several layers as-

sociated with the production steps. To make large scale designs possible, the elements of the circuit are composed of design components, such as resistor or MOS transistors. These design components are implemented in the software as parameterized cells, the so-called PCells. The purpose of a PCell is to programmatically draw shapes on the drawing layer to implement a complex component like a transistor. This is based on instance specific parameters specified by the circuit designer or an automatic design flow. Parameters may signify geometric information such as lengths and widths or electrical information such as capacitance and resistance or logical information such as terminals and connectivity.

A basic PCell is a set of functions, that implement the component by drawing the appropriate shapes based on the PCells parameter set. Whenever a parameter of the PCell is changed, the code is rerun to render the associated shapes.

While traditionally, for each new component, a custom PCell has been designed, the Extensible PCell Methodology (EPM) uses the fact, that to draw drawing functionally comparable components, like different transistor types, does not require different code, but only a different underlying configuration of the transistor. So the basic transistor code performs its work based on an abstraction of a transistor concept using a large set of variables for the abstraction elements. It is the task of the designer to implement actual transistors by specifying these values.

Because the set of parameters describing a transistor is large, but differs in a very systematic way between the different transistors one naturally wants to use a hierarchical approach to organizing the data. So here is, where the object oriented data model comes into play.

EPM is used via the `EpmDefClass` macro, which bears some resemblance to the CLOS `defclass` macro. It is used to set up the dataset for a PCell, by inheriting from existing classes and modifying or adding additional parameters. The `EpmDefClass` macro allows the evaluation of arbitrary Lisp code within the class definition via the `expr` operator.

```
(EpmDefClass classname ancestors
  slots)
```

FIG. 1: Syntax of an EPM class definition. A slot definition consists of a list containing the slot name and one or more property value pairs.

In Fig. 1 the basic syntax of an EPM class definition is shown. `ancestors` is a list of classes to inherit from, `slots` a list of slot definitions. An EPM slot definition differs from a CLOS slot definition substantially. A slot always holds some serialized form of data. A slot consists of a slotname and list of attributes. The attributes give meta data describing the slot.

Table I shows the attributes of an EPM slot. Besides the initial data stored, contained in the attribute

property	function
value	the value contained in the slot
min	the minimum value allowed for the data
max	the maximum value allowed for the data
pcellParam	the slot is a PCell parameter (a change of this slots value will cause a reevaluation of the PCell)
cdfParam	the slot will be display in the CDF form
constParam	the parameter is a constant
inheritanceRule	list of inheritance rules to use when merging slots

TABLE I: Some slot attributes, supported by the EPM system.

value, meta information about the data and its handling is given. Most important of those attributes is **inheritanceRule**, which allows the user to specify the inheritance rule that applies for this slot.

Upon inheritance, the attributes of slots inherited are combined with the definitions of the class. So an inheritance operation can just be to add or modify certain attributes to already existing slots in the base class. This is a very common operation, that the slots of a PCell type are given in the base class of that process, but only modified later on to create the actual component implementations.

rule	inheritance function
append	the inheriting class appends to the data (as a list)
error	an error is signaled
ignore	the parent overrides the child
prepend	the data of the inheriting class is prepended to the data (as a list)
max	the maximum of two numbers is taken
min	the minimum of two numbers is taken
merge	functionally merge two values
shadow	the child overrides the parent
union	create an unique list from the union of the parent and child lists
drc_merge	update the DRC rule list

TABLE II: The inheritance rules, supported by the EPM system.

However, more complex inheritance operations can be used as shown in table II. Numerical operations like **min** and **max** are supported as well a list operations like **append** and **union**. Finally there is the possibility of custom functions operating on the inheritance, like **drc_merge** which operates on the data in an application specific way. The inheritance attribute in a per-slot mapping of the attribute names to the type of inheritance to apply to each attribute. As the inheritance rules for a slot are stored as one of the slot attributes, they are themselves inheritable in the same way as the other slot attributes are. That not only means that they are inherited from all classes inheriting that slot, but also that

inheritance rules for the inheritance rules can be specified or modified by the inheriting class.

V. PRACTICAL EXAMPLE

In this section, a practical example of the usage of EPM is shown in a step by step process. It is shown, how EPM can be used to model a whole development kit tailored to a given production process. Of course, this example is minimized to show only the relevant part of the construction, not all details not relevant from a software point of view. A development kit is tied to a certain semiconductor production process. All design components to be used have to be customized for this process in respect with the principal physical properties of the process, as well as configured for the requirements of the actual design task to perform.

```
(EpmDefClass proc ()
  ((width
    value 2.0
    units "metric"
    min 1.0
    max 1e6)
  (metal1
    value "L_met1")
  (pdiff
    value "L_pd2")
  (ndiff
    value "L_nd1")
  ))
```

FIG. 2: The EPM code to set up an example process. The default values of common used variables are set.

Typically, for each process a base class is constructed from which all other classes inherit, as shown in Fig. 2. This base class sets up the relevant parameters of the process, like the names of the basic layers of the process to use and the principal attributes of all parameters to use, like here, that the width of each component is to be given in metric units, unless overridden in the derived classes. Furthermore, a default width of 2 microns and a possible value range between 1 micron and 1 millimetre is specified. However, these values can be overridden in any derived classes. Depending on the amount of configuration needed, setting up all components for a certain process might already be done to a great extend by the proper base class setup.

For each class of components like resistors and transistors, a base class then is created, inheriting from the processes base class. So all components share the definitions of the process. In the component base class, the customizations applying to all components of that type are made, like defining a component-specific layer.

Fig. 3 shows this for the resistor component class. The base class **res** defines a new slot to be used **reslayer**,

```

(EpmDefClass res (proc)
  ((width
    value 3.0
    min 1.5
    inheritanceRule (nil
                    value shadow
                    min max))
  (reslayer
    value "L_res1")))

(EpmDefClass res1 (res)
  ((width
    value 4.0
    min 2.5)))

(EpmDefClass hvres (res)
  ((width
    min 3.0)))

(EpmDefClass res2 (res hvres)
  ((width
    min 1.0)
  (reslayer
    value "L_res2")))

```

FIG. 3: Defining a set of resistors. The class `res` defines the basic resistor setup, the classes `res1` and `res2` actual PCells.

and a default width for all resistors. The actual PCell definition classes `res1` and `res2` override these values. The property `min` holds the minimum value for the width slot. The inheritance rules for the width slot are specified in the `inheritanceRule` property as a list. In this example it is specified, that for the value slot, the child overrides the parent value, but the maximum of all min specifications is used. So, while specifying 1.0 as the minimum for `width`, `res2` will still have a min value of 3.0, as it inherits from the class `hvres` and the inheritance rule for the min property demands that the maximum of all inherited values is taken.

In a more complex example, Fig. 4 shows a possible object hierarchy for MOS transistors. Besides the basic MOS class, the intermediate classes `pmos` and `nmos` are defined, which setup the attributes for p- and n-doped transistors. From these classes the actual devices (`nmos1` and `nmos2`) are derived.

The definition for the class `nmos` shows, how new slots can be introduced and initialized with values read from other slots. In this case, the default value of the slot `sdlayer` is set to the value of the slot `pdiff`. Though they are initialized with the same value, introducing a new slot, allows subclasses to redefine the value for the slot `sdlayer` without changing `pdiff`. Initializing `sdlayer` by referencing the value of `pdiff` has another important consequence in the case that a subclass changes the default value of `pdiff`, as `nmos2` does, because if no other statements about `sdlayer` are made, this means, that both layers get the given value.

```

(EpmDefClass mos (proc)
  ((width
    value 4.0)))

(EpmDefClass nmos (mos)
  ((sdlayer
    value (expr (EpmGet 'pdiff)))
  (gateLayer
    value (expr (EpmGet 'ndiff)))
  (layerRules
    value ((minSpacing 0.22 M1)
          (minSpacing 0.20 GC))
    inheritanceRule (nil value union))))

(EpmDefClass pmos (mos)
  ((width
    value 4.0)))

(EpmDefClass nmos1 (nmos)
  ((width
    value 4.0)
  (layerRules
    value ((minSpacing 0.33 M2)))))

(EpmDefClass nmos2 (nmos)
  ((width
    value 4.0)
  (pdiff
    value "L_pd3")))

```

FIG. 4: The code for defining a set of MOS transistors.

The slot `layerRules` gives an example of handling list type attributes via EPM. The value of this slot is the union of all definitions in all classes inherited from. So the `layerRule` given in `nmos1` is added to those given in the base `nmos` class.

VI. DISCUSSION

In this paper we have shown, how object orientation is a means to manage problems, which have an inherent hierarchical structure. Using object oriented programming techniques is a well established practise in software engineering. Applying these techniques to data organisation gives a powerful methodology to ease the phase of data definition. By reflecting the hierarchical nature of datasets for certain classes of problems, it increases the maintainability and helps ensuring data correctness. Furthermore, the objected orientated technologies are easier to apply to data than to code due to the fact that there are well defined operations on data. This allows for an automated handling of inheriting one slot from more than one class. So multiple inheritance becomes much more usefull on data object hierarchies than on program code.

A further extension of this methology could investigate the possibility to merge a traditional code object model like CLOS with the methods for data represented in this

paper. If successful, this could extend the applications of object orientation for software development.

- [1] Bjarne Stroustrup, The C++ programming language, Addison Wesley (1991)
- [2] Sonja E. Keene, Object-Oriented Programming in COMMON LISP, Addison Wesley (1989)
- [3] G. Kiczales, J. Rivieres, D. Bobrow, The Art of the Metaobject Protocol, MIT Press(1991)
- [4] Paul Graham, On Lisp, Prentice Hall (1993)