ACT Parameterization Framework

Alan Pavičić
AVL-AST Zagreb, Croatia
alan.pavicic@avl.com

Nikša Bosnić
AVL-AST Zagreb, Croatia
niksa.bosnic@avl.com

Abstract

ACT is a generic parameterization framework used in the development of applications for modeling and parameterization of internal combustion engines. It is developed in Guile. Its two main parts are *Ilm* core of object model built on top of Goops, and *Bee* editor environment providing UI. The core object model supports generic persistence of any object to database, type guardians for different slots, nameservices and object repositories. It also supports *addins*, additional modules which can change the behavior of the entire system as well as any of its parts (e.g. undo/redo functionality, dependencies between objects, event notification, ...). The editor environment for editing Ilm objects includes a library of basic editors, simple composite editors and generic editors. A grading system can be used to dynamically decide which registered editor class is the most appropriate for editing a particular object. Every Bee editor is an Ilm object itself. High level XML descriptions of data models and editors can be compiled to Scheme code defining Ilm classes and Bee editors.

Keywords Lisp, Scheme, MOP, data model, UI, parameterization

1. Introduction

We are working for AVL, a company producing software that simulates parts of internal combustion engines. Most products in our product line are structurally similar. They all consist of two main parts – a part which models and parameterizes some aspects of an engine and a part which actually calculates simulations (solver). Each solver is typically monolithic stand-alone process which reads custom formatted data files from input stream, and after (sometimes very lengthy) calculation stores the result of the simulation to some output stream to be additionally post-processed.

We will concentrate on the part which allows user to model parts of an engine and prepares the input data for solvers in the system.

Such a modeling and parameterizing subsystem needs to be able to define and edit some particular aspects of an engine (depending on the actual ability of the particular solver) and then run the solver. Previously, such a subsystem was implemented in such a way that particular solvers were run from different programs written in C++ which weren't mutually connected. This architecture was drastically slowing down adding new or changing existing aspects of the engine and every change in UI required programmer intervention and rebuilding of the whole application.

Sixth Workshop on Scheme and Functional Programming. September 24, 2005, Tallinn, Estonia.

Obviously, we needed more expressive and more efficient system. The first step in the implementation was the analysis of requirements.

In most cases, parameterization is not a very difficult task, because it can be reduced to a relatively small number of statically defined classes of objects which are being parameterized. Connections between such objects are typically trivial, or there are no connections at all. Similarly, editors for such objects can be hand written or just partially automatized.

Sometimes requirements on parameterization can be quite serious. In our case, we have a project where a large number of classes is in play, which are intensively changed during development of program or can be added to system after it has already been deployed.

Also, we have some non typical requirements on objects as they have to know how to persist and depersist themselves (save state to some unspecified medium, such as a file, an internet connection or a relational database, and be able to restore it later, e.g. after the program has been restarted).

Models described by our system can be quite complex themselves and dependencies between objects can be very specific (e.g. relations between mechanic parts of car engine).

Motivated by all of the above, we decided to create a modeling language which allows the same functionality to be added in different ways, depending on the estimation of application developer, rather than to create a rigid tool which should anticipate all possible requirements on classes and relations between them.

Apart from modeling requirements, there are also requirements for additional changes to the functionality of objects. Again, instead of anticipating all possible ways how the behavior of object could be changed, we rather open a way to change the behavior of any class or object during runtime. As we will see later, abstractions for changing metaproperties of an object we will call *addins*.

Similarly, the system has to be able to describe even particular editors for particular types of objects or any other elements of UI. The philosophy should be that the simple editors could be generated automatically and very quickly, but if the application programmer wants to add a very specialized editor for some class or family of classes, that should be possible too. Such approach would guarantee us both – fast development when possible, and tuning anything within the system when necessary.

Finally, it would be nice if even the application itself could be described as a regular object which behaves like the rest of the system.

Such a system, which couples all mentioned elements, would be a parameterization framework for rapid application development in any technical area, not necessarily just engine simulation.

The system described is specific enough that the object model of any typically used OO language (C++, Java, Python, ...) doesn't fit completely. Moreover, since we have requirements that classes can change their behavior (e.g. an object is able to log all changes

of its properties) no fixed object model would serve us completely, no matter how powerful it is.

Creation of such object model from scratch would be a long and expensive task.

Thus, we decided to use the meta object protocol (MOP)[9] which allows us to be independent from any predefined fixed object model and gives us freedom to change the object model on the fly as needed.

The most complete implementations of MOP can be found in Lisp systems, so Lisp was the most obvious choice from the beginning. Because, from a management perspective, the experiment of using Lisp could have failed, Lisp implementation had to be free to reduce possible losses. Because of high number of target platforms, implementation also had to be easy to port. An additional requirement on Lisp implementation was that the chosen implementation has to interface easily to C because of third party libraries we use (GTK+, expat, OpenGL, libuuid, ...).

We chose Guile as the Scheme implementation because it satisfies most of our needs. It is widely used, free and easily portable. It comes with Goops [3] – a complete CLOS-like implementation of MOP. Although it meets all of our requirements, decision to use it is still questionable[1] and implementation of the whole system shouldn't involve anything Guile specific on the conceptual level so everything should be easily portable to any Lisp which has a complete implementation of MOP.

Goops itself has some differences from CLOS, but it is still part of CLOS family. It has slots, generic functions, methods, metaclasses similar to CLOS but it lacks proper implementation of method combinations.

The object system we built upon Goops is named *Ilm*, and the editor system built upon Ilm is named *Bee*. *ACT* is the complete architecture for application development, which along Ilm and Bee contains *Xi* – an XML editor which allows application developers to simply draw definitions and layouts of classes and editors, *Xic* – a compiler from Xi XML formats to Ilm and Bee definitions, and some parts more specific to area of internal combustion engine simulation. Xi is created for the sake of more efficient application development and the fact that most of our application developers do not know Scheme.

2. Ilm

2.1 Ilm Basics

The basic idea of Ilm is to enrich Goops with new features, but to preserve the way the object system is used. That means there should be no difference between using Ilm classes and using classes which are instances of the default metaclass `<class>`.

From user's point of view the basic difference is that a class is defined using `define-ilm-class macro`, which is syntactically the same as `define-class` macro. The class defined in such a way has metaclass `<class-ilm>` and has class `<unique>` added to its list of superclasses. An additional difference is that slots, which do not have getter and setter names defined, will get standardized names for them (prefixing "get-" or "set-" and adding "!" at the end of setter name). We must enforce that access happens only through getters and setters because for some elements of the system to work, one may use only get/set functions to communicate with instances and should never work directly with `slot-ref` and `slot-set!` functions. Similarly, `#:init-keyword` is added if absent.

For example, the code:

```
(define-ilm-class <gas> ()
  specific-heat-capacity
  specific-heat-ratio
  dynamic-viscosity)
```

creates an Ilm class `<gas>` with fully defined slots.

Above definition is expanded to:

```
(define-class <gas> (<unique>)
  (specific-heat-capacity
  #:init-keyword #:specific-heat-capacity
  #:setter set-specific-heat-capacity!
  #:getter get-specific-heat-capacity)
  (specific-heat-ratio
  #:init-keyword #:specific-heat-ratio
  #:setter set-specific-heat-ratio!
  #:getter get-specific-heat-ratio)
  (dynamic-viscosity
  #:init-keyword #:dynamic-viscosity
  #:setter set-dynamic-viscosity!
  #:getter get-dynamic-viscosity)
  #:metaclass <class-ilm>)
```

The class `<unique>` has a single slot `uuid` which is set to unique 128 bit value during instance initialization. To generate that value, the libuuid library is used.

The second class essential for the system is class `<ref>` used for representing references. It is a simple Goops class which contains two slots – the slot `uuid` which keeps the uuid of the object the reference points to, and the slot `obj` which keeps the object itself. The value of the slot `obj` is `#f` if the target object is not loaded.

One of the basic requirements on the system is that every object must be persistable. Knowing that an object in its slot may contain any Scheme value including other objects or collections of objects, it is easy to imagine a situation where we have cycles in the reference graph (in fact this situation is very common when the model is complex).

Class `<ref>` is used for breaking the circularity during recursive persistence of objects. When another Ilm object is found during traversal through object's slots or compound values within a slot, we are persisting a reference to that other object using its uuid as the key rather than the found object itself.

When an object is instantiated or depersisted (loaded) it registers itself with the *object repository*. The object repository is a weak hash table whose keys are uuids of objects, and values are objects themselves. During depersistence, the system again recursively traverses through all object's slots and values. When a reference to an object is found, the system looks for matching a object in the repository and puts it to the proper place. If a matching object is not found (it is not depersisted yet), the system adds a broken link to the hash table and stores a location which should point to the missing object. Eventually, when the missing object is loaded all missing links are removed from the hash table and all pointers are set to their proper values. Such loading strategy enables lazy loading of instances, which is an advantage when large clusters of objects that do not have to reside in memory simultaneously need to be loaded. Obviously, the object repository has to be a weak hash table because if an object is not referenced by some other object (other than the repository itself), it should be collected.

2.2 Persistence

The serialization format of the persisted object does not depend on the database implementation and is always the same. That property allows easy implementation of persistence to a new medium.

Objects are always stored as S-expressions.

The basic writer for objects is the standard generic function `write`[2], with specialized methods for a few additional classes. The main change, with respect to standard write, is that `<ref>` and

[1] performance problems, bugs, module system deficiencies

[2] R^{5}RS [8] the `write` procedure becomes generic function after Goops is loaded

<unique> are written in custom syntax #,(instance ...) that stores the class name of the persisted instance and the keyword list of #:init-keyword value pairs. This syntax is the reason why every slot that needs to be persisted has to have #:init-keyword defined, and why Ilm will add one if omitted. Every database implementation has to provide a port used by write for storing the object.

Analogously, loading of object is implementation independent. Using define-reader-ctor from SRFI-10[6], #,(instance ...) syntax allows us to use the standard function read for reading from given port. If the class whose instance is being read is not yet present in memory, the system will look for its definition on the file system and load it before instantiating the object.

For example, a persisted instance of the above class <gas> could look like:

```
#,(instance <gas>
   #:uuid
   #,(uuid "c6e93456-fef8-44df-9738-d00df8926860")
   #:specific-heat-capacity
   #,(instance <ref>
      #:uuid
      #,(uuid "8426e7f7-1883-48a5-ab4b-43dcf94ba45d"))
   #:specific-heat-ratio
   #,(instance <ref>
      #:uuid
      #,(uuid "75cd206c-d03f-4288-ae1d-109a0e5360bd"))
   #:dynamic-viscosity
   #,(instance <ref>
      #:uuid
      #,(uuid "ac11af8e-0913-4a90-b16b-53b0e7903864")))
```

By default each bound slot with allocation type #:instance and which has #:init-keyword will be persisted. If we do not want to persist such slot, we can use keyword #:nopersist while defining the slot. If the value of the #:nopersist keyword is true, the slot is skipped.

The storage (database) where objects are persisted is named *object pool*, regardless of how it is implemented.

A valid implementation of an object pool is any library that satisfies the following requirements:

- it must invoke the standard read and write on its own ports while loading and saving an object
- it must support shallow loading and saving (i.e. implement load-object and write-object) using standard read and write
- it must support deep loading and saving (i.e. implement load-object-deep and write-object-deep)

Such definition of the object pool provides transparent scalability from trivial object pools (e.g. persistence to the clipboard used for copy/paste) to large databases.

It is recommended that object pool implementation indexes objects by uuid, while other indices are not required[3].

Most object pool implementations will have a symbolic name for their identification.

At the moment, three different object pool implementations exist:

- using the file system – The database name is the directory name, every object is in its own file named after object's uuid. Indexing is done by the file system.
- single file database – Used for embedding Ilm databases into other formats. The database name is the file name and the index is embedded in the file.
- Berkeley DB – Currently in the test phase. A hash table is used for indexing uuids.

Regardless of implementation, an object pool should be garbage collected periodically; otherwise dead objects can remain in it forever. The root set for the object pool garbage collector is the *name service*.

Object pools describe physical representation of the stored object. If we want to arrange objects in logical an hierarchy or we want to give a logical name to an object, we use <name-service>. <name-service> can be considered as analog to file system in Ilm world. A standard way for an application to get some particular object by its name from the the object pool, is using name service (using uuid is considered bad style since uuids should only be used internally and there is no guarantee the object will remain in the database if it is reachable only by uuid).

<name-service> is a standard Ilm class. Therefore, it can be persisted. Since it keeps references to other Ilm objects, placing another named instance of <name-service> within it creates a lower level in hierarchy in the logical sense. The root name service always has to exist and every object pool has to have a function for obtaining it. Typically, that function is named load-obj-from-named-source. If an object is not reachable from the root name service or some other named source it may be considered dead. <name-service> is a simple hash table.

2.3 Metaclasses, Aspects and their Applications

Every Ilm class is an instance of the metaclass <class-ilm>. <class-ilm> is derived from <class>. The initial reason for introducing additional metaclass to the base system is possibility to customize the initialize method for <class-ilm>, which allows us to change the behavior of the class we are defining before it is fully defined.

For example, Goops creates all getters and setters of a class as instances of <accessor-method> class. If we want to combine methods generated with getters and setters with some other methods, instances of <accessor-method> class are not sufficient because they do not support next-method (the form needed to combine methods). That is the reason why we replace all <accessor-method> instances that we get from slots with regular instances of <method> during instantiation of <class-ilm>. The implementation of newly created methods is taken from accesor methods.

The fact that getters and setters are regular methods is intensively used by addins.

We used the ability to modify a class during its creation to introduce several new keywords in slot definitions. #:getter-thunk and #:setter-thunk define post and pre processing procedures respectively which are used to modify default implementations of getters and setters. #:getter-thunk takes two arguments: the object whose getter is invoked and the value received from the default getter. #:getter-thunk that simply returns the value received from default implementation would be implemented as:

(lambda (obj val) val).

#:setter-thunk takes three arguments - the object, the new value and the procedure which would be invoked by default. A simple pass-through #:setter-thunk would be:

(lambda (obj val proc) (proc obj val)).

Slot definitions may omit thunks. One example when thunks should be used is automatic conversion of units (model internally uses SI units while values are provided in arbitrary unit system selected by user). In this case thunks would perform the unit conversion.

The last keyword we added for customizing slot definitions is #:type. It is used as type guardian for particular slot – if one tries

[3] The implementation of a query language is planned.

to assign to a slot a value of wrong type, a runtime exception is raised.

If a slot has both, `#:setter-thunk` and `#:type` keywords, the new value is first passed through the setter and than processed by the type checker. Types can be basic types like integers or strings, enumerated types (elements of a symbol list), other Ilm classes or compound types like type list and type union. A type union allows slot values of one of the specified types for that slot. A type list requires the value to be a list of instances of specified types for that slot, properly ordered. With such compound types, any recursive type can be described.

Example for canonical definition of list of integers without macro usage:

```
(define int-list (make <ilm:type-union>))
(set-types! int-list
            (list (make <ilm:nil>)
                  (make <ilm:type-list>
                        #:types (list
                                  (make <ilm:integer>)
                                  int-list))))
```

and redefinition of the class `<gas>` with a new type guarded slot `ints`:

```
(define-ilm-class <gas> ()
  ...
  (ints #:type int-list))
```

Types are used for better guarantee of correctness of program as well as to enhance introspection capabilities (used by generic Bee editors).

Like the ability to define classes separately from methods, it would be nice if parts of the same class could be defined separately. In practice, it is often a case that some property or a set of properties is defined later on, and that it is added to definitions of some already defined classes. For example, an engineer who describes a cylinder cares only about slots which are related to calculations in some particular simulation, but the class `<cylinder>` can have some additional properties not necessarily related to engine simulations (e.g. the name of the author and some documentation). Such sets of orthogonal properties of a class we call *aspects*. When an aspect is added to a class, new slots are introduced, but the class doesn't change its behavior in any other way. Every slot in the class stores which aspect introduced it. If a slot is supported by several different aspects, it contains a list of all those aspects. If different aspects introduce the same slot with incompatible settings (e.g. `#:init-value` is different), the system raises an error.

Information which aspects are supported by the class are stored in a slot of the metaclass `<class-ilm>`. The class and its slots can be queried and filtered by different aspects. The macro `define-class-aspect` is syntactically similar to the macro `define-ilm-class`, except that it takes the name of the aspect as its second argument. The implementation of aspects is basically redefinition of a class in a way that all already existing slots are kept and new slots are added, taking care about merging of properties of duplicate slots[4].

An example of a macro for adding an aspect to a class:

```
(define-syntax add-name-aspect
  (syntax-rules ()
    ((_ cls) (define-ilm-class-aspect cls #:naming
               (name #:init-values ""
                     #:type (make <ilm:string>))))))
```

and usage of that macro applied to the class `<gas>`:

```
(add-name-aspect <gas>)
```

[4] We are considering implementation of aspects using multiple inheritance that would enable specialization of methods by aspects.

Now `<gas>` has a new slot `name` and supports the naming aspect.

2.4 Addins

The basic behavior of objects (e.g. persistence) is always in the system. When we want to introduce some additional behavior of an object which for some reason (memory usage, speed, pure aesthetics, ...) doesn't need to exist for every class or object, we are introducing special types of modules, named *addins*. An addin can introduce a new behavior which cannot be described by the base system itself.

While aspects introduce new slots and don't change the behavior of the class, addins bring new functionality to existing methods.

Such enriching of the model with new a functionality we call injecting. Important features of addins are that they can be applied to any class or instance and that they can be combined. Number of additional addins which can be added to the base system is unlimited.

We will try to clarify addins through two examples – undo/redo and dependency addin.

A system which would keep track of all changes on all slots of every object all the time would at times be needlessly inefficient (e.g. when it is used by some calculation which is executed from a script where things like undo and redo make no sense). On the other hand, the ability to execute undo and redo actions on some objects and keeping track of all changes chronologically is quite helpful to application developers, who could use the object system without knowing how to implement undoing. Undo addin addresses exactly that issue. Even in an application that needs undo/redo functionality not all objects are undoable. All an application developer has to do to have undo/redo facility in his program is to declare which objects should be undoable or declare classes whose all instances should support that facility.

Injecting an addin means that a new class will appear in the system. The new class will be composed of two – the original class and a class which is introduced by the addin. Composition is done using multiple inheritance. The class introduced by the addin is typically an instance of some addin-specific metaclass, so the composed class will be an instance of the addin's metaclass too. Hence we can additionally customize the composed class in the `initialize` method of the addin's metaclass. In the undo/redo example, we are traversing through all setters, modifying them to register all changes on the global undo/redo stack and to invoke `next-method` which in turn invokes the original setter method, specialized for old class to which addin was injected. That is the reason why we had to convert all getters and setters from `<accessor-method>` to `<method>`. `undo` and `redo` functions are just executing closures stored on a global stack. Of course, changes are captured only when an object is changed through a setter and the object is an instance of an Ilm class with undo/redo addin injected.

If an application programmer knows in advance which addins should be used, and into which classes or objects they should be injected, he could use the composed class name – the name of class concatenated to the name of the injected addin. If we want to make an undoable instance of `<gas>`, we would create an instance of the class `<<undo><gas>>`, where `<undo>` is a the name of addin class.

If we want to inject an addin to an already instantiated object, after its class is composed with the addin, all we have to do is call `change-class` to the newly created class. Since the new class has superset of slots of the old one, all values within old slots will remain untouched. Instead of invoking old methods, such object will have more specialized methods for setters, which are created during composition of classes.

The purpose of the dependency addin is that slot values can be calculated from values of other slots (perhaps from another ob-

ject) by some user defined formula. If we make one slot dependent of other slots the connection will be stored in an instance of class `<dependency-descriptor>`, which also stores the dependency formula. Propagation of change is eager – immediately after some value is changed, the object knows whether it is dirty (needs updateing), but the calculation of the value is lazy and it calculates only parts it actually needs. The persistence of such cluster of objects will not calculate all dirty objects before they are stored. Rather, it will persist current in-memory state including `<dependency-descriptor>` objects. Same as in undo addin, everything what's happening during setting of the slot value and during reading from a slot is defined in `initialize` method of the dependency metaclass `<dep-mc>`. Original getters and setters are invoked using `next-method`.

When an object with an injected addin is loaded, the name of its class is recognized as composed class and after loading of class and addin, additional composition is performed.

For example:

```
(inject-addin-to-class <undo> <gas>)
```

will create a new class `<<undo><gas>>` whose instance will be persisted as:

```
#,(instance <<undo><gas>>
   #:uuid
   #,(uuid "c6e93456-fef8-44df-9738-d00df8926860")
   #:specific-heat-capacity
   #,(instance <ref>
      #:uuid
      #,(uuid "8426e7f7-1883-48a5-ab4b-43dcf94ba45d"))
   #:specific-heat-ratio
   #,(instance <ref>
      #:uuid
      #,(uuid "75cd206c-d03f-4288-ae1d-109a0e5360bd"))
   #:dynamic-viscosity
   #,(instance <ref>
      #:uuid
      #,(uuid "ac11af8e-0913-4a90-b16b-53b0e7903864"))
   #:ints (1 2 3)
   #:name "air")
```

To ejecte an addin can from an object, `change-class` to the original class can be invoked.

The list of possible addins is open ended. In addition to already described addins we implemented an event addin which makes an object notify its listeners when any of its slots change. Implementations of a locking addin which would replace proper setters with dummy setters and debug addin which is able to log all changes within the system are planned.

3. Bee

3.1 Bee Basics

For a complete solution of the parameterization problem, apart from the data model we also need *editors* – UI components dedicated to the interactive modification of objects. The part of our system addressing the task of editing Ilm objects is called *Bee*[5]. Altough Bee does not limit the choice of UI library, all currently created editors are implemented using GTK+ [4].

Every Bee editor is an Ilm object itself. That approach elegantly solves persistence of editors (i.e. UI state), dependencies between editors etc. Additionaly, it also enables creation of meta-editors (Bee editors designed for creation and modification of Bee editors)[6]. Furthermore, since Ilm permits modeling of the data model, Bee editors are also used for editing the data model itself i.e. Ilm class definitions.

An additional source of flexibility of Bee editors is a classical Lisp pattern where an editor accepts a procedure (commonly just a simple lambda expression) as a value of a parameter that specifies or specializes its behavior. Examples of problems solved in such a way are definition of arbitrary hierarchy in the generic tree editor (children of a tree node are returned by a procedure given as a parameter, effectively solving filtering and ordering also) and naming of an entity (name is generated depending on the context and/or translated to the given language).

Since an editor is fully defined just by defining six state-changing actions upon it, a short description of the life cycle of an editor (shown in figure 1) is necessary.

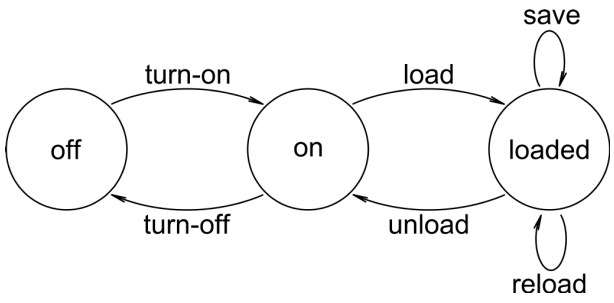

Figure 1. Editor state diagram

- The state *off* is the starting and the ending state. An editor in that state exists as an Ilm object but it still (or again) doesn't have any UI representation. This state is introduced to enable manipulation of properties of the editor which must be defined before the widget (or widget hierarchy) that makes up the editor's UI is created.
- In the state *on*, the static part of the editor's UI representation is created but it is not visible. The static part of UI representation is the part that can be created without knowing exactly which object will be edited and it includes at least the main widget of the editor. We can add an editor in this state to some parent widget and by doing so we can build UI to be shown later all at once.
- The state *loaded* is the "working" state of the editor. Before the editor can enter this state, the object to be edited must be set. UI representation exists in full and is visible, and the editor permits interactive modification of the object.

Actions *turn-on*, *turn-off*, *load* and *unload* switch states of an editor. All transitions shown on the state diagram are allowed (e.g. off-on-loaded-on-loaded-on-off) so the same editor can be used multiple times for editing (even editing different objects) without repeated construction and destruction of the static part of its UI representation.

Each action is implemented as a Goops method that can be invoked by the owner of the editor. Bee provides a simple embedded language[7] for defining editors:

- `specialize-ed-class` macro defines an editor class (using `define-ilm-class`) and overrides the default initial values for slots inherited from its base classes.
- Macros `define-turn-on`, `define-turn-off`, etc. simplify the definition of appropriate methods, provide error checking and ensure state consistency.

[5] short for "**B**ee is an **e**ditor **e**nvironment"

[6] This possibility is not employed in its full strength in the current implementation.

3.2 Basic Editors and Simple Composite Editors

Basic editors cover editing of ”atomic” objects and serve as building blocks for construction of complex editors. Typical examples of basic editors are editors for strings, numbers, enumerated values, Boolean values, tabular functions, physical quantities (a pair of a number and a unit from given unit group) etc. Although each basic editor must be manually coded[7], the embedded language described above greatly reduces the effort.

For example, the complete definition of an editor for real numbers is:

```
(specialize-ed-class <real-ed> (<gtk-ed>)
  (layout-hints '(#:hflexible #:small)))

(define-turn-on (ed <real-ed>)
  (set-widget! ed (make <gtk-entry>)))

(define-turn-off (ed <real-ed>))

(define-load (ed <real-ed>)
  (gtk-widget-set-sensitive (get-entry ed)
                            (not (read-only? ed)))
  (load-text ed))

(define-unload (ed <real-ed>)
  (gtk-entry-set-text (get-entry ed) ""))

(define-save (ed <real-ed>)
  (unless (read-only? ed)
    (set-obj! ed
              (string->real (gtk-entry-get-text
                             (get-entry ed))))))

(define-reload (ed <real-ed>)
  (load-text ed))

(define (load-text ed)
  (gtk-entry-set-text (get-entry ed)
                      (real->string (get-obj ed))))

(define get-entry get-widget)
```

Simple composite editors group several (often basic) editors into one whole. Layout creation algorithms have access to *layout hints*, a way for a child editor to express its properties regarding layout. While the current version of Bee includes only a simple single-column composite editor, a table composite editor is under development.

3.3 Grading

To any editor class we can attach one or more *graders* – procedures that, based on properties of the location we want to edit (e.g. allowed types of objects, type of the object currently stored at the location, read-only flag, ...), give a numerical measure of how appropriate an instance of the editor class would be for editing that location. That way we can make the decision about the most appropriate editor class dynamically, without the explicit knowledge about all editor classes in the system and their requirements.

One appropriate grader for the real number editor class could be registered as:

```
(registry-add-type-grader
 (lambda (type)
   (and (or (is-a? type <ilm:real>) (eq? type <real>))
        (cons <real-ed> 11))))
```

A later call to a query function such as `registry-grade-type` would include a pair of the editor class `<real-ed>` and the grade 11 in the returned list if the type given satifies the above condition.

[7] as opposed to automatically generated

By convention, more specific editors are given higher grades. A non-specific ”last-resort” editor intended primarily for use by application developers can be used for any location but gets a low grade. On the other hand, an editor created for a specific narrow category of objects (like `<gas-ed>` below) gets much higher grade, but under more selective conditions.

3.4 Generic Editors

The concept of graders opens a door towards *generic editors*. Generic editors use simple composite editors as containers and layout managers for editors created according to the results of the grading of parts[8] of the object being edited. The simplicity of this process enhances scalability with respect to the number of classes in the data model and the number of editor classes in the system, along with resilience regarding data model changes. Furthermore, generic editors enable work on the data as soon as the data model is finished or even during its development.

For example, figure 2 depicts an instance of `<uni-ed>`, a generic editor that uses the described grading and the single-column composite editor, editing an instance of the class `<gas>` defined with the appropriate slot type information. `<uni-ed>` grades each slot of the given object, selects the editor class with the highest grade if any, and adds an instance of the selected editor class to a single-column composite editor serving as a container and layout manager.

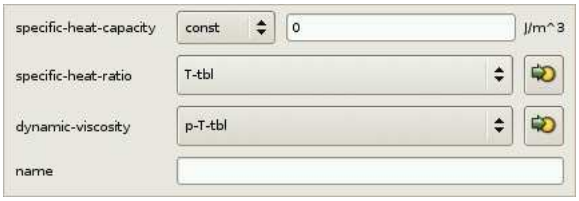

Figure 2. A generic editor

With some minimal specialization generic editors can often replace complex editors built manually by gradual composition of basic editors:

```
(specialize-ed-class <specific-heat-capacity-ed>
  (<multi-type-ed>)
  (slot-namer (make-alist-namer
               '((const . "Constant")
                 (T-tbl . "Table (T)")
                 (p-T-tbl . "Table (p,T)")))))
(registry-add-class-type-grader
  <specific-heat-capacity>
  <specific-heat-capacity-ed> 13)

;;;; omitting similar specialization code for specific
;;;; heat ratio and dynamic viscosity editors

(specialize-ed-class <gas-ed> (<uni-ed>)
  (heading "Gas")
  (slot-namer
   (make-alist-namer
    '((name . "Name")
      (specific-heat-capacity ."Specific Heat Capacity")
      (specific-heat-ratio . "Specific Heat Ratio")
      (dynamic-viscosity . "Dynamic Viscosity")))))
(registry-add-class-type-grader <gas> <gas-ed> 13)
```

The specialized generic editor `<gas-ed>` is shown in figure 3.

[8] typically non-virtual instance slots

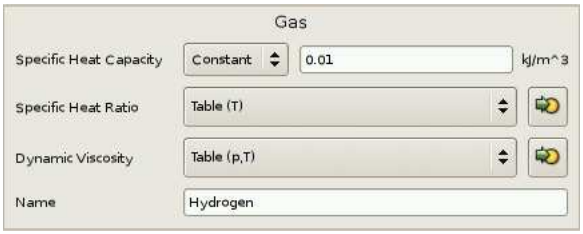

Figure 3. A specialized generic editor

4. Related Work

ASL's [10] components Adam and Eve2 treat problems that are similar to those treated by Ilm and Bee, respectively. We avoid the multiple language approach (C++ for the implementation of libraries, AEL for data and dependency expressions, AVM for the interpretive runtime execution) by using Scheme for all parameterization purposes.

Cells [1] and Cells-Gtk [2] combined also provide a flexible Lisp based parameterization framework.

5. Conclusion

The initial predevelopment experiment was successful and after a short additional development phase we are about to start with deployment, proving once again that Lisp and MOP should be considered in commercial programming at least equally to C++ or Java. Problems we encountered using Lisp weren't of conceptual nature and mostly were related to the selection of a good implementation that covers our specific requirements.

References

[1] *Cells*. http://common-lisp.net/project/cells

[2] *Cells-Gtk*. http://common-lisp.net/project/cells-gtk

[3] *Goops*. http://www.gnu.org/software/guile/docs/goops

[4] *GTK+*. http://www.gtk.org

[5] *Guile*. http://www.gnu.org/software/guile

[6] *SRFI-10*. http://srfi.schemers.org/srfi-10/srfi-10.html

[7] Paul Graham. *On Lisp*. Prentice Hall, 1993.

[8] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). *Revised(5) Report on the Algorithmic Language Scheme*. http://www.schemers.org/Documents/Standards/R5RS

[9] Gregor Kiczales, Jim des Rivières, Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[10] Sean Parent, Foster Brereton. *Overview of Adobe Source Libraries*. http://opensource.adobe.com/group__asl__overview.html