Lua/P — A Repository Language For Flexible Software Engineering Environments

Stephan Herrmann
Technische Universität Berlin, FR 5–6, Franklinstr. 28/29, 10587 Berlin, Germany
phone: +49 30 314 73174 – email: stephan@cs.tu-berlin.de

Abstract
Ongoing development and combination of methods and tools for software development call for software engineering environments (SEE) with ever changing functionality. Also the integration of operative support for the software development process remains a major challenge. A good SEE design has to combine a high level of integration with great flexibility towards evolving methods and tools as well as adaptability towards different kinds of development projects. We have developed PIROL as a generic SEE demonstrating that an **executable meta model** may play a key role in combining integration and flexibility. We coin the notion of a **repository language** to denote a domain specific language for the domain of repository based meta models. We introduce Lua/P as PIROL's repository language with some non-standard properties and show how this language contributes to the desired properties of the environment.

Keywords: Repository, meta-modeling, domain-specific language, tool integration

1. Introduction

Although there is plenty of software tools, many development projects still lack a suitable, state-of-the-art software engineering environment (SEE). The reuse of SEE components, their composability and configurability still falls behind the demands of concrete development projects. In spite of all commonality, projects differ considerably in multiple dimensions. This is for a large part a matter of different workflows and can be modeled in notions of documents, states, persons, resources and schedules. Two factors add to these problems: (a) software development is a matter of very intense communication and (b) a very tight semantical integration of different (sub-)documents should be supported by the environment across tool borders.

It is of course possible to hand craft a specialized SEE for any project just using existing techniques. Only the effort needed is far too high. Thus the uniqueness of software development projects calls for techniques that allow to build a specific SEE from components in just a fraction of the time needed so far.

Many standards and techniques have emerged for solving single concerns of integration. Some focus on techniques for communication between tools (cf. CORBA[1], COM[2] etc.) others tend towards meta formats for data exchange (XML) or even standardize parts of the environment's meta model (XMI). Some approaches are very general techniques that add no special solutions for SEEs; others help under the precondition of a fixed set of notations which is not appropriate for many fields of software engineering where still new formalisms and combinations thereof are widely explored (cf. [3]).

In earlier work [4] we have presented the vision of using an executable object-oriented meta model as the central concept for a tightly integrated yet configurable SEE. Advantages include (a) a semantical enrichment of an otherwise purely syntactic data model, (b) tool independent implementation of process and framework integration (cf. [5]) and (c) adaptability of the meta model through subclassing and scripting. In this paper we elaborate on certain requirements towards this meta model. We show how data modeling can be extended with efficient implementation of very fine grained data structures and elaborate techniques for preserving consistency in the presence of multiple views on shared data. The driving force is always the desire to reconcile a high level of integration with clear modularity allowing flexible configuration of a concrete environment from existing parts.

We point out the importance of the language that is used for meta modeling. Such a language (the meta meta model) will be called a *repository language* throughout this paper. It is a domain specific language for the task of defining meta models as a basis for and as integrative glue of SEEs. The repository language has to blend in with the techniques used for persistence and communication. We will present current work on the repository language Lua/P that is part of the PIROL SEE.

The following section will give an overview of PIROL and the systems it is built upon. Sect. 3 shows how Lua/P encapsulates the underlying repository. Sect. 4 sketches the communicational context. Sect. 5 discusses issues of safeguarding consistency in the repository. Sect. 6 tackles the question of granularity with regard to data modeling and processing. Sect. 7 motivates a disciplined technique of class migration. Sect. 8 wraps up with some examples of applying Lua/P. Just a few implementation details are collected in Sect. 9.

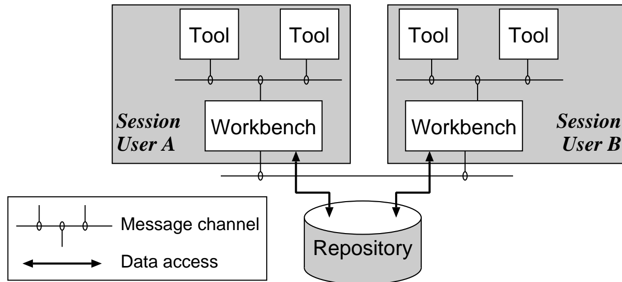

Fig. 1. Architecture of PIROL

2. A Generic SEE: PIROL

PIROL (Project Integrating Reference Object Library)[4] is an SEE designed for supporting all five dimensions of integration as defined in the ECMA reference model[5]. In this section we will give an architectural overview of the system.
PIROL is built according to a three–tier architecture as illustrated in Fig. 1:
1. Data storage is provided by a *repository* which is accessible only through a dedicated server process.
2. For each user a *workbench* defines his or her working context. The workbench caches accesses to the repository and provides private visibility of the user's current work. Communication between the workbench and the repository is based on the mechanisms provided by the repository. Also, messages may be sent to other workbenches.
3. *Tools* are connected to a user's workbench by means of a special messaging protocol. Tools only communicate with the workbench, never directly with the repository. Support is given for the implementation of new tools and also for the integration of existing ones.

The focus of this paper lies on the workbench and on Lua/P, PIROL's repository language which is implemented by an interpreter as part of the workbench. However, before presenting Lua/P we will briefly introduce two systems on which all this is based.

2.1. H–PCTE

PIROL uses a system called H–PCTE[6] as its repository (cf. Fig. 1). H–PCTE is an implementation of the ISO standard PCTE[7]. Its central component is the *object management system* (OMS), that defines a non–standard database system. PCTE closely adheres to extended entity relationship models. Thus the main elements are entities (objects), relationships (links) and attributes. The latter can be attached to either objects or links. The database is non–standard in that the primary access is not via tables and key attributes but by navigation along links.

H–PCTE is implemented (and continuously being improved and maintained) by the group "Praktische Informatik", at Siegen University[8]. A special focus of the design of H–PCTE was the performance issue in conjunction with fine grained data modeling. H–PCTE spells "high performance PCTE".

2.2. Lua

Lua[9] is developed at PUC-Rio, Brazil. It is titled an "extensible extension language". The notion extension language is to say that Lua is well-suited for integration into a given application in order to provide a high level facility for configurability, macro programming and the like.

Within our context the extensibility of Lua itself is of greater importance. Lua is adapted very easily to specific needs yielding a great variety of derivatives that could be called a *family of languages*. Lua is a (basically) interpreted language, which comes as a library that can be linked to applications. It is extensible by two mechanisms:

- C functions may be registered as Lua functions.
- The behavior of the interpreter can be modified by the Lua code itself. This is done by redefining standard constructs like reading or writing a field of an object.

At the time we started using Lua for PIROL this second mechanism was called "fallbacks" and operated globally for all kinds of objects. The PIROL development required to use fallbacks for very different purposes (depending on the kind of object). This contributed to an improvement of the mechanism, which is now called *tag methods* and can differentiate kinds of objects by a tag that is attached to each object.

By means of these extension mechanisms, we extended the host language Lua to encapsulate H–PCTE and add object orientation to Lua, as well as all those features that qualify it as a specialized repository language. Lua/P spells "Lua for PIROL".

3. Encapsulating the Repository

Lua/P encapsulates the repository in a way that combines the following properties:
1. Lua/P objects are persistent without the need of explicit read and write operations. Reading is lazily performed when dereferencing a link, writing occurs immediately upon every attribute assignment or object creation.
2. Lua/P is used to define all data types in the repository and adds behavior (methods) to repository classes.
3. Many functions from the large PCTE-API are encapsulated by core classes of the meta model written in Lua/P. There is no need to access PCTE directly.
4. Lua/P itself is an evolving language that gives specialized support for the most common concerns in SEEs. The following sections will focus on the concerns of consistency (Sect. 5) and fine grained meta modeling (Sect. 6).

Properties (1) and (2) mainly define Lua/P + H–PCTE as an OODBMS. Note however, that Lua/P is not meant to compete with standard OODBMS, but it combines

features of an object oriented database language with specialized features for programming SEEs.

We had to consider some subtleties in the type system of Lua/P as compared to PCTE: The type system of PCTE knows types of attributes (string, integer, boolean), objects and links, which are straight forwardly mapped to basic types, classes and object references in Lua/P. Lua/P, however, (a) encapsulates 1:n links by a List class and (b) adds lists of basic values and lists of tuples. These additions of course need to pay attention to efficiency, so some of these types are packed into one binary attribute in PCTE, or make use of link attributes, which otherwise have no representation in object oriented languages. Only tuples with more than one object component cannot benefit from such optimizations. They have to be represented by additional PCTE objects. Sect. 6 shows how structured data types can be added to this type system with minimal runtime overhead.

In addition to persistent objects, Lua/P also allows to declare classes or attributes as *transient*, which is important where the creation of temporary objects as repository objects would impose an unnecessary performance penalty. Clients, however, need not know about this distinction because persistent and transient objects are handled in a uniform way.

4. How Tools Access the Workbench

It has been shown how Lua/P encapsulates the underlying OMS. This section briefly shows the interface through which tools can operate on repository objects. This defines the context from which execution of Lua/P code is triggered and motivates why the Lua/P interpreter must support change propagation between tools.

The channel used for all communication between workbench and tools is a messaging facility with a multicast protocol. This facility is based on the package MSG from the FIELD environment[10] with significant modifications, that are not discussed here. Messaging is implemented by a server and a client library such that all clients can easily establish a socket connection to the server. The server is responsible for message dispatching. Workbench and tools are clients of the message server, which is in general transparent to both as they communicate with each other.

In PIROL six types of messages exist by which tools may request services from the workbench:

query: Read the value of one attribute (simple or complex).

query list: Query a detail about a list attribute. Four sub-functions exist:
`length()`, `item(index)`, `search(value)`, `filter(constraints)`

set: Assign a simple attribute value.

set list: Lists can only be modified by these sub-functions:
`append(value)`, `replace(index,value)`, `insert(index,value)`, `delete(index)`.

execute: Execute a Lua/P method, passing any number of arguments.

create: Create a new repository object. This may include a call to a Lua/P creation method.

Additionally the workbench broadcasts all attribute changes to the message channel. This is a very important feature of the Lua/P interpreter, because PIROL is designed for supporting multiple views, which obviously need to be kept consistent by means of some mechanism of change propagation. It is left to each tool to register a pattern for each object or attribute which is displayed by the tool. The tool then receives all relevant update messages and may update its display accordingly. List updates are sent as incremental changes (reflecting all those `append, replace ...` operations). For simple attributes the new value is passed with the update messages.

In PIROL tools can generally be implemented in any programming language. Currently most tools are implemented in Java.

A class library exists for *Java* that encapsulates the messaging library and provides proxy classes for all classes of the meta model core. Proxy classes typically have `get_`*xx* and `set_`*xx* methods for each attribute. Lists are encapsulated by a Java class, which automatically observes all changes of the corresponding list in the repository. It may register additional observers that propagate changes within the tool. Proxy classes have static methods wrapping the creation of a repository object. Method calls are directly delegated to the workbench. The main limitation of the Java client library is the lack of multiple inheritance in Java. So a *complete* mapping from Lua/P to Java is not possible.

A client library using *Lua* may exactly imitate the behavior of the workbench. No `get_`*xx* or `set_`*xx* methods are needed. Not even proxy classes are needed for this library, since they can be build on the fly from meta information available from the workbench. Currently however no tool is using this technique.

5. Consistency

PIROL emphasizes the role of multiple views within a repository based environment. This imposes obligations to safeguard the consistency of all views and their representations as objects in the repository. Consistency has to be ensured at least on two levels: when regarding tools as view–control components according to a model–view–control architecture, different views need to be kept consistent by means of change propagation as mentioned in the previous section (cf. also [11]). A more semantical understanding of consistency concerns invariants and semantical constraints describing interdependencies between different objects/attributes in the repository. In this section we will focus on the latter aspect of consistency, but we will also relate this aspect to techniques

Event	Arguments	Description
Simple attributes:		
assign	`object, value`	assignment of `value` to the resp. attribute of `object`.
get	`object`	retrieve the resp. attribute from `object`.
List attributes:		
adding	`list, index, value`	`value` is being added to `list` at position `index`.
removing	`list, index, value`	`value` is being removed from `list` at position `index`.
append *remove* ...	} all regular list functions	

Fig. 2. Events for attribute guards

of change propagation.

5.1. Guarded Attributes

Lua/P has been presented as a language for data definition and manipulation. It is generally possible to set any attribute of any object to any value, as long as access permissions and type correctness are observed. This is very comfortable for most cases, but sometimes this weak encapsulation is not sufficient. For achieving a better encapsulation, Lua/P provides the mechanism of *guarded attributes* which allow to implement further constraints. Different applications of this technique are possible:

1. Consistency constraints of the meta model might require a change of one attribute to be propagated in terms of changing also some other attributes/ objects.
2. Other constraints might allow only specific changes, in which case assigning a wrong value should either throw an error or just do nothing.
3. Some attributes may represent something outside the Lua/P interpreter. Changing their value should produce a side–effect by calling some low-level interface.

Examples for the three kinds of constraints are:

1. Classes `CLASS` and `ROUTINE` from the PIROL meta model both have a flag `is_deferred`[1]. The constraint is: a class that has at least one deferred routine is itself deferred. Both flags are implemented as guarded attributes. When a routine is set to deferred, the corresponding class is also marked deferred, when a class is set to not deferred, it is ensured that also all routines are not deferred.
2. In class `WORKBENCH` an attribute `current_group` defines, on behalf of which group the current user is working. He or she may adopt another group simply by assigning that group to `current_group`. Given only this, every user could possibly gain the rights of *any group* by a simple assignment. However, implementing `current_group` as guarded attribute allows additional checking. Only groups, of which the user is a member are allowed. Violating this restriction generates a warning and the assignment is refused.
3. In fact the previous example already hints at a third application of guarded attributes. The attribute `current_group` is not only documentation but effectively defines the permissions of the current user. The effective access rights are always the combination of the user's personal rights and those of his or her current group. This is achieved by calling the PCTE function `pcte_adopt_group()` each time the `current_group` has successfully been assigned.

[1] Deferred stands for 'not fully implemented', 'abstract'. This is one of a set of notions which we borrowed from the nomenclature of Eiffel[12].

Definition of Guarded Attributes. Guarded attributes are declared and used just like others. Only a set of guard functions is added that is used for certain events. The syntax of a full guard definition is:

```
AttributeAccess Class.Attribute {
  Event = function(Arguments)...end,
  ...
}
```

This is for simple attributes. For list attributes the keyword `ListAccess` is used instead. Fig. 2 shows the events that can be (re)defined, Fig. 3 on the next page shows an example. Here only the *assign* event is overridden by a function that ensures the consistency of the flag `is_deferred` from our example (1) given above. `raw_assign` is the actual assignment function to be used within an *assign* function.

It should be mentioned that guarded attributes are mainly a matter of convenience. We could have required that all attributes be accessed only by means of explicit access functions (even within the same class!), which would, however, unnecessarily clutter the code and should actually only be used, where needed. That's why in Lua/P no reason exists to ever call a function like `get_attribute` or `set_attribute`. On the client side it is important that regular and guarded attributes are used in the same way. This guarantees that common tools like generic browsers are able to exploit guarded attributes in a meaningful way, without knowing about their guards.

5.2. Derived Attributes

The previous section showed how to ensure consistency in case of interdependencies between different objects/attributes. It is certainly a good idea to minimize the necessity of such consistency constraints by avoiding redundant data in the repository wherever possible.

```
Class ROUTINE {
   attributes={
      is_deferred : Boolean,                              -- Declaration
      ...
   },...
}
...
AttributeAccess ROUTINE.is_deferred {
   assign =                                                     -- Event
      function (routine, flag)                          -- Guard function
         if flag == routine.is_deferred then
            return
         end
         raw_assign(routine, "is_deferred", flag)
         if flag then
            local class = routine:get_class()
            class.is_deferred = flag
         end
      end
}
...
local routine = ...
routine.is_deferred = true        -- Application triggering the guard
```

Fig. 3. Guard for attribute `ROUTINE.is_deferred`

```
Class ROUTINE {
   inherit FEATURE,
   attributes={
      signature : Derived(String),                        -- Declaration
      arguments : List(ENTITY),
      ...
   }
}
...
function ROUTINE:derive_signature ()  -- Derivation Function
   local args = self.arguments:foldl ( "(",
      function (arg, pre)
         if not pre == "(" then pre = pre..", " end
         return pre..arg.signature        -- read another Derived
      end)                                                   Attribute
   args = args..")"
   if self.type then
      return self.name..args..": "..self.type.name
   else
      return self.name..args
   end
end
```

Fig. 4. Derived attribute `ROUTINE.signature`

PIROLs fine grained meta model is a major step towards absence of redundancy. Another means is the mechanism of derived attributes.

As an example of derived attributes, consider the signatures of routines. Names, arguments and result types of routines are kept persistently either as direct attribute (name) or using separate repository objects of types `ENTITY` (arguments) and `TYPE` (result type). It should on the other hand still be possible to query the signature of a routine (encoded as a human readable string) with just one query. Pre-assuming a redundant *attribute* `signature` should be avoided for the sake of consistency, a *function* `ROUTINE:get_signature()` would be a good starting point, but it has a great disadvantage as compared to attributes. Only attributes allow to register observers that inform a client about all changes of the corresponding value. Function results may become invalid without further notice.

Derived attributes now combine the best of both worlds. They are free of redundancy because they are not persistent but computed when needed. Syntactically a derived attribute is an attribute and most importantly the workbench broadcasts all changes of the values of derived attributes for which a tool holds an observer. Fig. 4 shows how a derived attribute is declared using the type constructor `Derived` and how a derivation function is attached as `derive_`*xx*`()`.[2]

6. Fine Grained Meta Models

Fine grained data modeling is a powerful means for a tight integration of tools that are to share as much information as possible. Of course an object oriented meta model could very well be used to decompose a document all the way down to single identifiers and symbols. This technique is however not usable for SEEs. A prominent approach to fine grained data modeling for SEEs has been standardized as extension of PCTE[13]. Unfortunately no tool vendor ever really implemented this standard due to tremendous performance problems that should be expected. Database technology in fact imposes quite strict limits on the number of objects that can be accessed efficiently when eg. loading a document.

Quite a different lesson can be learned from the area of compiler construction and related tools. Such tools rely on a set of types that represent all constructs of a (programming) language in a tree or DAG structure, called *abstract syntax*. The definition of these types and many transformations are much more compact and perhaps more elegant when using a functional programming language rather than an object oriented one. For this reason a previous instantiation of PIROL [3] that was targeted at processing formal specifications based on their abstract syntax used the programming language Pizza[14]. We made good experiences with Pizza's combination of object oriented and functional techniques. In this setting the bottleneck was the serialization of Pizza objects. Serialization, which was used to write units of the abstract syntax as binary blocks into the repository, again imposed performance problems on the system.

In response to this experience, Lua/P was extended by some new features: The types needed for an abstract syntax or similar structures can be defined as *term grammars*. Terms as values of these types can be handled and stored efficiently by the workbench. Allowing term types for attribute declarations yields a smooth integration of medium grained objects and very fine grained terms. Finally a touch of functional programming in Lua/P allows concise implementations of algorithms over terms.

6.1. Term Grammars

Terms are tree structures whose leaves are simple values or terminal symbols. Simple values are strings, integers, boolean or subtypes thereof. Lua/P provides four

[2] ".." is the Lua operator for string concatenation. For the function `foldl()` cf. Sect. 6.2.

```
  Grammar {"EXPRESSION";
(1)   expr    = one_of{value, binexp, unexp, ifexp},
(2)   binexp  = {{e1=expr}, {operator=binop}, {e2=expr}},
(3)   unexp   = {{operator=unop}, expr},
(4)   ifexp   = {{condition=expr}, {then_exp=expr},
                        {else_exp=expr}, '?'},
(5)   binop   = subtype_of{STRING},
(6)   unop    = one_of_const{{uplus='+'}, {umin='-'}},
(7)   value   = one_of{INT, BOOL, varappl},
(8)   varappl = subtype_of{STRING},
(9)   vallist = {value, '*'},
  }
```

Fig. 5. Grammar EXPRESSION.

kinds of type rules (the LHS of each rule is a type):

subtype_of The LHS type can be used wherever the RHS type is required. It has the same structure.

one_of The LHS type has alternatives that are listed here. The alternatives still have to be defined.

one_of_const Similar to the above but the alternatives are terminal symbols given by their representation.

production The LHS type is a tuple of the types listed at the RHS. Production rules have no keyword.

Fig. 5 gives an example grammar defining a simple expression language. The names `e1, operator` and `e2` as defined in rule (2) are selectors for the components of an `expr` term. The second component in rule (3) is not named, so the type name `expr` is also used as selector. The `'?'` in rule (4) specifies that the last component (`else_exp`) is optional. The `value` in rule (9) may occur zero to many times (denoted by `'*'`). Elements of such a list can only be accessed by their numerical index. Finally the whole grammar is given a name in order to make it a selectable name space.
Each type defined in a grammar can be used for attribute declarations as in

```
Class SIMPLE_FUNCTION {
  inherit ROUTINE;
  attributes={
    value : EXPRESSION.expr
  }
}
```

When storing such values to the repository a compact yet type safe binary encoding is used to pack complex terms into a single attribute of PCTE. The effect is that for Lua/P each (partial) term is a well defined entity and terms are structures built according to strict type rules. The system's performance however does not suffer from such very fine grained data modeling, because no additional PCTE objects are created.

6.2. A Touch of Functional Programming

Being able to define term types by grammars as shown above we left pure object oriented techniques. The new types can only be exploited effectively if we also add the appropriate functions for *handling* terms. Fortunately, Lua already provides the basic mechanism for functional programming. In Lua a function is a value that can be assigned to variables and can be passed as function argument or result. Additionally *function closures* are supported which by so-called upvalues provide a clean solution to the problem of variable scoping as it occurs in nested functions.[15]

Next to pure higher orderedness a special merit of many functional languages is their support for *pattern matching*. In Lua/P this is done by a function `t_switch` which matches a given term against a list of type patterns. Patterns are given by `t_case` branches. In the simple case, each pattern specifies a type and a function that should be executed, if the term is conform to that type. The function is called with the term as only argument. In addition to the top-level type a pattern may also give a list of types to which the subterms must conform. If such a pattern is matched, the subterms are passed as distinct arguments to the function. When the string *representation* of a term is desired this conversion can be automated by prepending the `@` operator to the type pattern. Finally a `t_otherwise` branch may provide a default function, that is used if no pattern is matched.

```
function expr2string (t)
   return t_switch(t,
(1)     t_case('@value',
          function (val)
             return val
          end),
(2)     t_case({'expr', '@binop', 'expr'},
          function (e1, op, e2)
             return('('..expr2string(e1)..op..expr2string(e2)..')')
          end),
(3)     t_case('unexp', {'@unop', 'expr'},
          function (op, expr)
             return('('..op..expr2string(expr)..')')
          end),
(4)     t_case('vallist',
          function (list)
             return '{'..
                    list:foldl('',
                          function (v, col)
                             if col ~= '' then col=col..', ' end
                             return col..v.repr
                          end)..
                    '}'
          end),
        t_otherwise(
          function () return '?' end)
   )
end
```

Fig. 6. Using pattern matching for a simple pretty printer.

See Fig. 6 for a simple pretty-printing function for expressions as defined in Fig. 5. Note, that in object-oriented programming the standard technique for this problem would be to apply the visitor pattern, introducing far more overhead than the more functional approach.

The first branch matches subtypes of `value`, the next branch matches any term consisting of an expression,

a binary operator and another expression. Branch (3) combines matching of top-level type (`unexp`) and structure (unary operator and expression). All operators and values are passed by their representation (use of `@`). Expressions are passed as terms. Branch (4) again is a simple match by type. It shows an application of the `foldl` function, which is borrowed from ML[16]. We introduced `foldl` to Lua/P as one of the most general higher order functions, that iterates over a list, collecting the results through a second argument (`col`). In this example the effect resembles a smarter mapconcat function: the representation of all list elements are concatenated using `', '` as a separator except for the first element.

7. Upgrading

In most object oriented languages the type of an object is an unchangeable property. In PIROL it has reasons of methodology that it should be possible to create an object with an unspecific type which later-on, as more information is gathered, is converted according to a more precise type. This may even iterate along several steps, such that the object's type stepwise becomes more and more specific. Because references to such an object must remain valid, it is important that

- the object identity is not changed and
- the object remains conform to the typing of all incoming references.

It is the first demand, that leads to *upgrading* as a special language feature rather than a mere copy method. The second demand restricts upgrading to a type change from a superclass to one of its subclasses. (For type conversions, or "migration", cf. eg. [17]).

Regarding the consistency of attribute values, upgrading is very similar to object creation. At creation time a creation method puts the object into a sound state with respect to its attribute values. When upgrading an object new attributes may be added, which are initialized by an upgrade method. Lua/P also introduces a hook called `upgrade_pre` which may be used to decide if upgrading should be allowed or disallowed. This way a class may put forward constraints on the state of objects that shall be upgraded.

At the user interface level upgrading is available through an operation "`insert as ...`". A selected unspecific object may eg. be inserted into a class diagram *as class* or *as attribute*.

8. Application of Lua/P

This section gives examples for parts of an environment that may be implemented in Lua/P and benefit from execution within the workbench.

8.1. Process Modeling

Many projects have taken efforts on formalizing the dynamics and constraints of software development processes. Using Lua/P for this task is elegant because all relevant entities are equally at hand and can be handled in a uniform manner. A process model in Lua/P may at the same time refer to a `SUBSYSTEM` that is being developed including all contained `CLASS`es as well as the `PERSON` who is responsibly for delivering these items in a certain `STATE` using certain (time–, hardware– ...) `RESOURCE`s etc.

As a simple example we present a state machine that manages the progress of development items along a chain of document states like *busy*, *proposed* (for publishing), *published* etc. Each artifact carries a reference to an object of class `STATE`. `STATE`s are connected to *next states* by `TRANSITION`s. Transitions in turn have a `GUARD` and an `ACTION`. In the most simple implementation `GUARD` objects specify which persons/groups are allowed to perform the corresponding transition and an `ACTION` object specifies how the access permissions of an object shall be changed.

This state machine can be adapted on two levels. First, the concrete set of states and transitions is a graph of objects that can be initialized by a Lua/P script once for each project. Second, the algorithms of guards and actions can easily be redefined by subclasses of `GUARD` and `ACTION`.

8.2. Common Services

Class `WORKBENCH` is central for the PIROL meta model. For each user an object of this class defines the working context that is available to all tools. This object contains eg.

- a list of tools that are installed and configured,
- a reference to the current project, group (cf. Sect. 5.1) etc.
- a simple clipboard.

Based on this information, services are implemented like selecting all tools that are available for editing a given object, providing lists of recipients to whom messages can be sent (within the group, project, company ...). Other services that are implemented in Lua/P are version control and access control.

Uniform Context Menus. All these services are really helpful only if they are easily performed at the user interface. An example of a flexible integration of services into the interface of all tools is the mechanism of *workbench provided context menus*. In addition to any local context menu as it may be needed for operating a tool, a submenu `Workbench ▷` is always provided. Concerning the selected object, this menu is dynamically put together by the workbench. The menu definition is kept as a set of transient objects within the workbench. The tool only reads the menu definition and displays the menu. Upon selection of one menu option the corresponding command (function closure) is executed within the workbench. This way no tool needs to know about any common service in particular, but all services provided by the workbench can be exported easily to all

tools. Inserting a new service (like the state machine shown above) or changing the configuration of a service (like adding a new transition to the state machine) has immediate effect on the context menu within all tools.

9. Implementation Issues

So far we have presented the current state of Lua/P. We did not develop this language from scratch nor in one single step. Lua/P evolved from Lua in several incremental cycles. We would like to give some hints on how Lua made this development fairly easy.

Lua [9], [15] is in fact an interpreter *framework* in the sense, that it is an application with many *hot spots*, through which a programmer may add and modify the application's behavior. In Lua these hot spots are those tag methods that have been introduced in Sect. 2.2. By means of tag methods, a simple assignment like `obj1.att2=obj3.att4` can be redirected to invoke one tag method for retrieving the value of attribute `att4` from a repository object `obj3` and another tag method for storing this value in attribute `att2` of another repository object `obj1`. Both tag methods make use of calls to functions from the PCTE API that for this purpose have been lifted from C to Lua. As seen from the Lua interpreter, repository objects are opaque handles, whose semantics are defined solely by tag methods. A similar technique is used for term values (cf. Sect. 6) which are efficiently implemented in C (which allows easy interfacing also with other programming languages).

All access functions, that are stored with individual Lua/P classes, are elegantly stored in function tables using Lua *closures*, that contain all necessary context information besides the function proper.

Another example for tag methods is the invocation of creation methods: the Lua/P statement `obj=CLASS1:make(arg)` denotes the creation of an object of `CLASS1` using the creation method `make` for initialization. This is implemented in Lua as follows: retrieving field `make` from `CLASS1` yields a method-*descriptor*. When trying to execute this descriptor as a function, a tag method (for event `function`) is called, which will find out, that no target object exists but one must be created prior to invoking the actual method `make`.

The use of associative arrays (cf. [9]) allows to write down all additions to the language in a usable, descriptive syntax without ever touching the implementation of the Lua parser. Only tiny changes have been made to the code examples in this paper, in order to give it a more standard appearance. Such modifications can easily be handled by a simple pre–processor.

Note, that aside from Lua/P no other language is required: no data definition language, because this is just one role of Lua/P, and no scripting language for whichever purpose, because Lua/P can also be used for scripts that automate any repeated tasks.

10. Conclusion

Several languages and systems exist, that are partially related to the work presented in this paper. PCTE's DDL is a pure data definition languages. ODMG-ODL[18] and CORBA-IDL[1] define object interfaces in terms of attributes and methods. As early as 1987, Garlan [19] defines different languages for defining different kinds of views. His *basic views* define the structure and behavior of objects in the database, *dynamic views* can be compared to derived attributes, with the restriction, that their results must be sets of objects. Pizza[14] is a programming language that combines object oriented and functional techniques. Within the UniForM workbench[20] concurrent Haskell is used for encapsulating repository objects and tools and implementing tool communication. Derived attributes are supported by notations ranging from Object–Z up to UML. OPM[21] is a specialized DBMS that implements derived attributes.

None of these systems defines a repository language as comprehensive as Lua/P. We should clarify that PIROL is not a production environment competing with industrial tools. Until now, some well known concerns like fine-tuned transaction management remain incomplete in PIROL, but by means of PIROL and Lua/P we are able to demonstrate, which abstractions and mechanisms may be covered by a repository language in order to enhance modularity and integration of an SEE.

We have shown that Lua/P is well-suited as a data definition language and at the same time lifts the repository types to a comfortable object oriented programming language. Fine-tuned type mappings and the introduction of transient attributes and objects provide for optimizations as they minimize the number of repository objects. Using term grammars for the definition of very fine grained data further improves the performance of the system, because terms can be packed into a single attribute.

Next to efficiency, Lua/P provides two mechanisms for preserving data consistency. Derived attributes help to avoid redundancy. Guarded attributes may either restrict attribute changes or operationally enforce consistency by propagating changes to other attributes/objects. Additionally, guarded attributes may lift properties of the underlying repository to Lua/P.

Aside from data modeling, Lua/P can be used for implementing the dynamics of repository objects using methods in the common object oriented sense. Addition of functional techniques enhances Lua/P's capability for transformations over complex structures like abstract syntax.

Finally, Lua/P closes the gap between tools and the repository. All persistent objects are accessible via a specialized messaging facility, which helps to keep visualizations up-to-date by broadcasting all relevant changes.

The process model and all common services that are implemented in Lua/P are independent from (but still

contribute to) any integrated tool. Modularity of the resulting SEE allows for configuration and adaptation at different levels: (1) Selection of methods and notations has impact on the product related part of the meta model and on the selection of tools. (2) Also adaptation of the process model or common services requires a few specific additions to the meta model. (3) Tailoring an environment that has been constructed by the above steps for a concrete project is mainly a task of writing simple Lua/P scripts.

Of course the crucial part in putting together an SEE from diverse components remains implementation, adaptation and integration of new and existing tools. We have made good experiences with tools providing different levels of openness. This ranges from tools specifically written for PIROL up-to monolithic tools with only insufficient interfaces[3]. We have good results especially with a graphical editor that could be adapted at the source code level. A clearly delimited adapter layer suffices for a very close integration.

Most importantly an object oriented meta model written in Lua/P greatly fosters the separation of concerns of different tools and the workbench. Thus flexibility of the SEE PIROL is essentially founded on the specific design of its repository language Lua/P.

10.1. Current and Future Work

We are currently working on specialized language constructs for defining connectors that help to integrate existing tools by automating a mapping between two mismatching meta models [22]. This approach also overcomes a major drawback of *basic views* according to [19]: in some cases it is necessary to multiply instantiate a certain view with regard to a common base structure.

Allowing wildcards in attribute names for guards (cf. Sect. 5.1) will allow to use guards as advice in the style of AOP[23], [24].

It will be easy to lift PCTE's distinction between association (usually: existence links) and object composition (composition links) to Lua/P. This powerful feature is unknown to common object oriented programming language.

References

[1] "The Common Object Request Broker: Architecture and Specification, revision 2.1," TC Document formal/97.9.1, OMG, 1997.

[2] S. Willliams and C. Kindel, "The component object model: A technical overview," Tech. Rep., Microsoft Corporation, available from `http://www.microsoft.com`, 1994.

[3] R. Buessow, W. Grieskamp, W. Heicking, and S. Herrmann, "An open environment for the integration of heterogeneous modelling techniques and tools," in *Proceedings of the International Workshop on Current Trends in Applied Formal Methods*. October 1998, number 1641 in LNCS, Springer.

[4] B. Groth, S. Herrmann, S. Jähnichen, and W. Koch, "Project Integrating Reference Object Library (PIROL): An object–oriented multiple–view SEE," in *Proc. of SEE'95*, Noordwijkerhout, Holland, April 1995, ACM–Press, Malcolm S. Verrall.

[5] "Reference Model for Frameworks of Software Engineering Environments – ECMA TR/55 3rd edition," Tech. Rep., European Computer Manufacturers Association (ECMA), June 1993.

[6] Udo Kelter, "H–PCTE — a high–performance object management system for system development environments," in *Proc. COMPSAC '92*, Chicago, Illinois, Sept. 1992, pp. 45–50.

[7] "ISO/IEC 13719-1: Portable Common Tool Environment (PCTE)," Abstract specification, International Organization for Standardization (ISO), 1995.

[8] U. Kelter, "Einführung in H-PCTE," Skriptum, Fachgruppe Praktische Informatik, FB Elektrotechnik und Informatik, Uni Siegen, 1998.

[9] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, "Lua—an extensible extension language," *Software: Practice and Experience*, vol. 26, no. 6, pp. 635–652, 1996.

[10] Steven P. Reiss, "Connecting tools using message passing in the FIELD environment," *IEEE Software*, vol. 7, no. 4, pp. 57–66, July 1990.

[11] D. Platz and U. Kelter, "Konsistenzerhaltung von Fensterinhalten in Software–Entwicklungsumgebungen," *Informatik Forschung und Entwicklung*, vol. 12, no. 4, pp. 196–205, 1997.

[12] Bertrand Meyer, *Eiffel: The Language*, Prentice Hall International, New York, 1992.

[13] "Amendment 1 to ISO/IEC 13719-1: Fine-grain object extensions," Tech. Rep., International Organization for Standardization (ISO), 1995.

[14] M. Odersky and P. Wadler, "Pizza into Java: Translating theory into practice," in *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[15] L. H. de Figueiredo R. Ierusalimschy and W. Celes, *Reference manual of the programming language Lua 3.2*, `http://www.tecgraf.puc-rio.br/lua/manual`.

[16] L. C. Paulson, *ML for the working programmer*, Cambridge University Press, 2nd edition edition, 1996.

[17] J. Su, "Dynamic constraints and object migration," in *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, Eds. 1991, pp. 233–242, Morgan Kaufmann.

[18] R.G.G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade, Eds., *The Object Data Standard: ODMG 2.0*, Morgan Kaufmann, 1997.

[19] David Garlan, *Views for Tools in Integrated Environments*, Ph.D. thesis, Carnegie Mellon University, May 1987.

[20] C. Lüth, E. W. Karlsen, Kolyang, S. Westmeier, and B. Wolff, "HOL-Z in the UniForM-workench – a case study in tool integration for Z," in *Proceedings of the 11th International Meeting of Z Users (ZUM'98)*, J. P. Bowen, A. Fett, and M. G. Hinchey, Eds., Berlin, 1998, number 1493 in LNCS, pp. 116–134, Springer.

[21] I.A. Chen, A.S. Kosky, V.M. Markowitz, and E. Szeto, "Constructing and maintaining scientific database views," in *Proc. of the 9th Conference on Scientific and Statistical Database Management*, August 1997.

[22] S. Herrmann and M. Mezini, "Dynamic view connectors for separating concerns in software engineering environments," in *Procs. of MDSOC workshop at ICSE 2000*.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, and J. Irwin, "Aspect Oriented Programming," in *Proceedings of ECOOP '97*, 1997, number 1241 in LNCS, pp. 220–243.

[24] *AspectJ Language Specification*, available from `http://aspectj.org`, Aug 1999.