Design, Use and Implementation of SPELL, a language for Software Process Modeling and Evolution

Reidar Conradi¹, M. Letizia Jaccheri², Cristina Mazzi³, Minh Ngoc Nguyen¹, Amund Aansten¹

¹ Norwegian Institute of Technology, Trondheim, Norway
² Politecnico di Torino, Italy
³ University of Pisa, Italy

Abstract. SPELL is a language for software process modeling based on a structurally object-oriented data-model with relations. It extends the underlying versioned EPOSDB, and is based on Prolog with full object-orientation, concurrency, persistency, distribution, and tool invocation facilities. SPELL can express multiple level of abstraction/composition of process information. A process model is a set of types to describe activities (tasks), products, tools and projects (management information). Task instantiation can be partly automatized. SPELL also provides a platform for modeling and experimenting with software meta-activities to model, analyze, and support software processes.

1 Introduction

Software Process Modeling (PM) and associated environments have drawn increased attention within the Software Engineering community. The need for flexible customization and evolution has lead to very dynamic process modeling formalisms, e.g. utilizing an object-oriented (OO) framework.

The paper will present SPELL, an object-oriented process modeling language for specifying, interpreting, and evolving process models. The structure of this paper is as follows: Section 2 introduces some PM concepts, and describes some requirements for a PM language and modeling framework. Section 3 describes SPELL on top of the EPOS kernel software engineering environment. Section 4 gives examples of process submodels formulated in SPELL. Section 5 describes the SPELL implementation: representation and tools. Some conclusions are given in section 6.

2 Requirements

A process model and its formalism must consider the requirements and properties of the real process. Some relevant characteristics are:

- Modularization, e.g. by hierarchical models or reusable model fragments.
– Abstraction, and possibilities for gradual Specialization (cf. PMi phases below).
– Evolution and Customization:
  A useful model must evolve to reflect changes in the business and to absorb improvements, and it must be customisable [LB85].
– Formalization, and thus support for automated analysis and assessment.
– Monitoring and feedback mechanisms, to assist above assessment and evolution.
– Clarity and Orthogonality, so that a small set of well-defined concepts can be freely combined.
– Understandability by humans, e.g. through an external graphical notation.

A process model and associated formalisms have at least four submodels:

– An executable activity model (or task model), to express both simple and aggregate activities. The available activity formalisms fall into four main categories:
  - Descriptive or rule/trigger-based (MARVEL [KF87] and ADELE2 [BEM91]).
  - Network-based (MELMAC [DG90], Extended Petri Nets [GMMP91]),
  - Imperative or programmatic Process Modeling Language, PML, usually interpreted (APPL/A [JHO90], IPSE 2.5 [War89]).
  - Hybrids (EPOS [C89]).
All the corresponding formalisms are rather low-level, with insufficient facilities for design, structuring and customization/evolution of models and their rule bases. The last point involves meta-processes, see below.
– A product model to express (passive) data, being manipulated by activities. An object-oriented ER model is often used.
  Note, that the product is evolved by activities, often driven by the product structure. The activities are themselves evolving and persistent artifacts, i.e. “products” being operated upon by meta-processes. Indeed, the model is itself a manipulatable product.
– A tool model to describe tools and their architecture. This can partly be expressed by the activity model, embedding a tool as an activity “envelope”.
– A project (or organisational) model to structure and control activities, and their executable resources. The organisational structure may reflect product decomposition. Common artifacts are humans (e.g. modeled by roles), machine resources, overall project constraints, team coordination, and work delegation.

Thus, a process model may cover an entire application, not only its “active parts”.

The following six meta-process phases can be identified, and need to be supported by methods, formalisms and tools [JLC92]:

1. Establishing a PM Framework.
   This consists of formalisms, pre-defined models, methods and tools.
2. Eliciting/designing an informal process model
   This is done with input from the application domain(s).
3. Establishing/analyzing a **formal, generic process model**
   Thus, this meta-phase involves browsing, editing, and analyzing a library of
   reusable model fragments (often types).
4. Compiling/customizing a **specific process model**
   Here the selected (sub)types etc. are refined and bound.
5. **Instantiation** of concrete subprocesses.
   Task subnetworks are made according to the actual products, schedules,
   tools, and human resources.
6. **Execution/monitoring** of processes, plus feedback.
   This mainly consists of guided or enforced use of application tools, interpreting
   a specific process model.

In PM systems with *dynamic binding*, like EPOS, the meta-process phases 3–6
are incremental and never-ending. Note, that SEI have introduced a somewhat
different taxonomy for meta-processes [FH92].

### 3 The EPOS System and its SPELL

We present the EPOS system layering and EPOS PM rationale, and then the
object-oriented **SPELL** (Software Process EvoLutionary Language) for defining,
interpreting and evolving process models.

#### 3.1 EPOS System Layering

The EPOS system is layered as follows:

1. A **client-server EPOSDB, with uniform (“change-oriented”) versioning** [L^+ 89]. It offers a **structurally object-oriented** EPOS-OER data
   model and its DDL to define entity and (binary) relation types^4. Entities
   have a unique identity (OID), and both entities and relationships can have
   scalar attributes with Simula-like inheritance. Entities can also have longfield
   attributes to describe external files.
   A few system-defined types are available, including the root **Entity** and
   **Relationship** types. All (meta-)type information is stored as **TypeDescrs**.
   Thus, type information is explicitly manipulable and versionable, and new
   types can be incrementally added.
   EPOSDB is implemented in C and with a client interface in Prolog, our DML.
   Extents of subtypes are **subsets** of those of the supertypes, and queries are
   performed accordingly. Database operations are performed in the context
   of a nested and long transaction, coupled to a project and with a check-out
   workspace. The current version context determines the “visible” sub-
   database.

---

^4 The EPOS-OER entity types are close to classes in OMG terminology [omg00]).
   This incorporates the DDL and DML above, and adds type-level attributes
   and instance/type-level procedures and triggers. The type-level information
   is used to assist type evolution. Before this layer 2 was defined (spring 1992),
   layer 3 below had to rely on ad-hoc semantics to define PM-specific type
   information.
   SPELL is implemented by a Translator/Editor and Code Interpreter, using
   the underlying EPOSDB.

3. An EPOS PM tasking framework, using SPELL.
   This consists of special type-attributes and type-procedures in a pre-defined
   TaskEntity type to facilitate tasking, and thus to define activity models.
   Task execution relies on the Execution Manager and the task network
   instantiation on the Planner (next section). All PM tools in layers 2 and 3 use
   the XPCE User Interface package.

4. General and project-specific types and instances, including application
   tools.
   Products and their task networks display both “vertical” aggregation (de-
   composition) and “horizontal” dependencies (chaining), connected to tools
   and human resources. Some standard types are available here.

3.2 The EPOS PM Rationale: Some Comments

EPOS combines separate AI rules with a more static (but automatically generated) task network. This hybrid modeling provides powerful and flexible behavior of tasks. Tasks represent coarse-grained and possibly long-lived activities, while procedure calls (possibly augmented by triggers) represent fine-grained and shorter activities.

The PM binding semantics has undergone extensive updates in object-oriented direction over the last year. The goal is to achieve more high-level and cleaner modeling, as well as meta-level instrumentation and evolution.

There are three axes of structuring and variability in EPOS process models:
— Vertical structuring by subtyping and other project specialization;
   i.e. the activity rule base is hierarchical.
— Horizontal evolution inside projects and versioning between projects.
— Combined vertical/horizontal, e.g. by dynamic binding semantics.

3.3 SPELL – the high-level EPOS PML, extending the EPOS-OEER

SPELL is an object-oriented extension of the underlying DDL and DML to better describe task dynamics and model evolution. The EPOS Process Framework consists of SPELL, some pre-defined types and meta-types, and available PM tools instrumented by type-level information. This subsection will describe the object-oriented SPELL extensions, consisting of procedures, triggers, type-level properties, and associated semantics. Instance- and type-level information are specified in two distinct sections inside a type, as shown in later examples. The
pre-defined TaskEntity and DataEntity subtypes of Entity are the roots of the task type and product type hierarchy, respectively.

Using SPELL, a mailbox facility and primitives for communication and coordination between distributed applications (transactions) have been defined; but is not treated here.

Procedures and Triggers The basic DML is a set of free-standing Prolog predicates. We need to structure this DML and the application programs in an object-oriented framework. We also need type-level procedures to instrument the Planner and PM Manager.

A SPELL example of two procedures and a trigger for the root Entity type is shown in Figure 1 (the syntax will not be formally explained; Prolog is used internally). The two Upd_E and Qry_E procedures are general Write and Read access-functions.

ENTITY_TYPE Entity {
  INSTANCE_LEVEL % No type-level definitions here.
  ATTRIBUTES
    Created : Time = ...;
    Written : Time = ...;
    ...
  PROCEDURES
    Upd_E(Self,O:OID, T:TID, [Attr-list]) = ...Written:=current-time.
    Qry_E(<Same as above>) = ...
  TRIGGERS
    ON-PROC = Upd_E  WHEN = AFTER
    COND = <If waiting tasks>  ACTION = <Put these into a "hot" list>;
} % Entity-type (root type)

Fig. 1. The Entity type, modified with procedures and a trigger.

Procedures are defaultly inherited by subtyping unless redefined, as in Smalltalk. A procedure body may be redefined, but its interface must be stable (no Cardelli-like rules). Indeed, the set of procedures and triggers on a type can change dynamically! Observe, for instance, that check-out and check-in procedures can be nicely defined in DataEntity subtypes to perform data conversions suited for database-external workspaces.

Triggers react to events, being calls of an ON-PROC procedure. The detailed semantics of trigger binding will not be described here [CPA+91]. In Figure 1, each Upd_E call will be instrumented by a tailing trigger. This can be used to selectively alert the Execution Manager to reevaluate the dynamic PRE-CONDITIONs of waiting tasks (see TaskEntity in Sec. 4.1).
Type-level attributes and their inheritance rules These attributes can also be of domains Prog (same as for procedure bodies and trigger ACTION bodies) and Pred (also used for COND part of triggers), and are primarily used by task types. Special inheritance semantics can be defined by setting Inner = redef (local redefinition), append (e.g. AND conjunction), or inner (INNER concatenation à la Simula-67). The latter two are only meaningful for the Pred and Prog domains respectively. Default values can be redefined in subtypes.

4 Process modeling with SPELL

We shall describe how SPELL is used to describe the four different submodels for activities, products, tools and projects. For the latter submodel, the transaction mechanism of the underlying EPOSDB is also employed.

4.1 Activity model

An activity model is a set of task types, plus associated relation types. A task type expresses knowledge about a basic or composite development step and its agents (tools, human roles). The root task type TaskEntity type is illustrated by Figure 2.

Some comments on the type-level properties for tasks:

- TYPE LEVEL ATTRIBUTES, general comment:
  Observe the domains Pred, Prog and String, mostly interpreted by the PM facilities.

- TYPE LEVEL PROCEDURES, general comment:
  Most of these procedures concern creation, update and deletion of types, as well as creation of instances. A more detailed explanation of their behavior is given in section 5.4. The instance-level procedure i_convert can be used to refine the type of an instance, and start, stop, and restart to control task execution.
  The two special variables Self0 and Self1 have the standard semantics of self.

- PRE- and POST-CONDITIONs:
  PRE_STATIC, PRE_DYNAMIC, POST_STATIC, and POST_DYNAMIC are formulas in first-order predicate logic that predicate on input objects state. Static conditions are evaluated by the Planner for task instantiation, and dynamic ones by the Execution Manager at execution time.
  The dynamic POST-CONDITION enables simple exception handling upon false, but is not described here.

- CODE:
  This defines the actions performed when the task is executed, i.e. each time its dynamic PRE-condition is true. The CODE is written in an appropriate sequential language, e.g. Prolog.
ENTITY_TYPE TaskEntity:Entity {
  INSTANCE_LEVEL
  ATTRIBUTES
    Exec-state: . . := . .; % For task implementation.
    Task-state: String := 'Created'; % Similarly.
  PROCEDURES % See under PM Manager.
    inconvert(SelfO:OID, NewT:TID) = . . ;
    i_delete(SelfO:OID) = . . ;
    start(SelfO:OID) = . . ;
    restart(SelfO:OID) = . . ;
    stop(SelfO:OID) = . . ;
  TYPE_LEVEL
  ATTRIBUTES
    PRE_STATIC (Inher=append) Pred = true
    PRE_DYNAMIC (Inher=append) Pred = true
    CODE (Inher=inner) Pred = . . INNER . .
    POST_STATIC (Inher=append) Pred = true
    POST_DYNAMIC (Inher=append) Pred = true
    FORMALS (Inher=redef) String = 'in:$DataEntity(read) ⇒
         out:$DataEntity(read/write)'
    DECOMPOSITION (Inher=redef) String = 'REPERTOIRE(TaskEntity)'
    EXECUTOR (Inher=redef) String = '<Logical tool name, e.g. CC>'
  PROCEDURES
    t_create(SelfT:TID, ChildT:TID, <Defined properties>) = . . ; % Called by SPELL Translator in PM Manager.
    t_delete(SelfT:TID) = . . ; % See PM Manager.
    t_change(...) = . . ; % Same parameters as t_create.
    i_create(SelfT:TID) = . . ; % Normal NEW generator.
    subgoals(...) = . . ; % Subtyped, used by Planner.
} % TaskEntity-type

Fig. 2. The TaskEntity type
- DECOMPOSITION:
  This constrains the task types of new, decomposed subtasks of a given task, i.e. it constrains the standard SubTasks relation type. It is mainly used by the Planner.

- FORMALS:
  This constrains the product types of actual in/out-parameters of a given task, i.e. it constrains the standard GenInputs and GenOutputs relation types. It is again mainly used by the Planner.

The FORMALS specification is technically given as
'in:$DataEntity(read) => out:$DataEntity(read/write)'
where '$' means a list. The names in and out are used for accessing the actual parameters. The next keyword is used for improving some static checking on semantics: in this way it may also be possible to read the output and to update the input.

- EXECUTOR:
  This attribute refers to the logical name of the tool that should execute the task, and corresponds to a “soft” type-instance relationship.

4.2 Product Model

An product model is a set of product (i.e. DataEntity) subtypes, plus associated relation types. The product model is not oriented towards any specific development method or programming language.

The type Part is used to describe any software part, with possibilities for instance-level aggregation. The Family type uniformly describes a collection of subsystems (a library) as well as a single module. The Interface and Body types assume that program modules have a public interface (possibly “faked” or derived by a tool), and a more private body. Figure 3 shows the definition of the Csource type for “c” files.

In each data type, instance-level procedures define operations to be executed on instances of this type and its subtypes. Such procedures are either called directly by the CODE part a task or indirectly by procedures. As for task types, the type-level procedures regulate creation of instances and type evolution.

4.3 Tool Model

The prototypical tool type is a Deriver task subtype. It serves as an envelope around a batch-oriented OS-tool, such as a compiler, link-editor, text formatter etc. We can envisage a Compiler subtype, with a CC-Compiler subtype etc. Interactive OS-tools are similarly modeled as Interactor subtypes.

The CODE of such task types specify preparing, activating, controlling and recording a tool activation, with proper INNER subtyping. As explained in sub-section 4.1, the EXECUTOR type-level attribute indicates which “symbolic” OS-tool should be invoked.
ENTITY Csource:Body {

INSTANCE_LEVEL
ATTIBUTES % No local instance-level attributes defined.

PROCEDURES
  compile(self, <h-files>, Context):-
    call Planner to build derivation graph, start(Context).
  UpdE(...)-: <body: calling root UpdE proc.>

TRIGGERS
  ON-PROC = UpdE WHEN = AFTER ACTION = compile().
  % Added trigger on UpdE procedure.

TYPE_LEVEL
ATTIBUTES
  LegalSuffixes String = '[c]'

PROCEDURES % No local type-level procedures defined.
} % Csource-type

Fig. 3. The Csource type

The OS-tools themselves are represented by passive, versioned objects of type Executable. They function as ad-hoc “external procedures”, whose instances are invoked by e.g. Deriver tasks. Note, that such OS-tools cannot be expressed as traditional object-oriented procedures in product types. This is because their bodies can be shared by several types, and because they are independently produced and versioned by the surrounding OS. Conceptually, there is a “N:1 relationship” between Deriver subtypes and OS-tool instances.

OS-tool switches cannot easily be modeled by traditional parameters, due to varying number and special semantics. E.g., some switches will cause extra or entirely different outputs to be produced, or may assume other inputs. That is, different deriver “variants” must be defined with appropriate FORMALS\(^5\). Our solution is to provide the default tool switches from the global CurrProject. "Crucial" tool switches, affecting the FORMALS, are contained in a constant DeriverSwitches attribute, whose default value may be subtyped.

4.4 Product Model for organisational information

Administrative and organisational information is modeled by Project types. Projects are special tasks coupled to database transactions. They control general cooperation within and between teams. E.g. the propagation rules between cooperating transactions is regulated by instrumentable task types [CM91]. A subproject can change old information “inherited” from the superproject (e.g. type versioning), as well as add new information (e.g. by subtyping). There are p.t. no sub-schemas, so the normal version mechanism must be used. Thus, EPOS has powerful basic mechanisms for structuring knowledge bases.

\(^5\) Indeed, the Unix C compiler may be used both as a pre-processor, compiler, linker, assembler . . . and with different FORMALS within these categories!
Project is a TaskEntity subtype to emphasize the productive aspect. A project has a limited lifespan, and may consist of subprojects. The project information includes the following information:

- An intentional config-description for the current EPOSDB transaction, with a description of local workspace organization. It may be supplemented by a traditional database view with rights and capabilities.
- Project-specific information, such as task types to express work procedures and task templates, project-pervasive attributes, ...
- A coarse OS description: specific or low level OS-tool information, such as file bindings, environment flags, default tool set with DefaultSwitches, e.g., such as -I <lib name> for the CC compiler, ...
- Subprojects, and allocation of various resources – as in ISTAR.

5 Implementation

5.1 The SPELL Internal Representation

To represent task and product models, the meta-type EntTD is extended with two meta-subtypes, i.e. DataTD and TaskTD. DataEntity and TaskEntity are created as instances of these meta-types.

DataTD and TaskTD define the necessary attributes and relations to store the type properties for activity model and product model types.

Type-level procedures and type-level attribute definitions (not values) should have been stored in separate meta-types. However, instead of making one meta-type per declared type as in Smalltalk, we have stored this information in a separate section of the types themselves, as explained in figure 5.1.

Task types may define attributes for its instances following the subtype semantics described in section 3.3. In addition, both data and tasks can define behavioral properties, i.e. procedures and triggers. These are represented as instances of separate objects, called procedure and trigger respectively, and are connected to the type through appropriate relations.

Since type-level attributes store more information than instance-level ones, e.g. inheritance semantics, TaskTD and DataTD attributes are represented as instances of TypeAttr, which is a subtype of Attr. Some type-level attributes of tasks, (FORMALS, DECOMPOSITION and EXECUTOR are mapped into a instances of relations.

5.2 The SPELL interpreter: call proc

The access and bind semantics of procedures, triggers and of attributes is implemented by one predicate: call proc (Called, Caller, ProcName, InputParms, OutputParms).

self is a default for an empty Caller. The Execution Manager will provide tasking.
### type with metatype info

<table>
<thead>
<tr>
<th>Instance Id</th>
<th>Id1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
<td>TaskTD</td>
</tr>
<tr>
<td><strong>Type Name</strong></td>
<td>TaskEntity</td>
</tr>
<tr>
<td><strong>Instance Attributes</strong></td>
<td>TaskState</td>
</tr>
<tr>
<td><strong>Instance Procedures</strong></td>
<td>i_convert</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td><strong>Type Attributes</strong></td>
<td>formals</td>
</tr>
<tr>
<td></td>
<td>pre_static</td>
</tr>
<tr>
<td></td>
<td>code</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td><strong>Type Procedures</strong></td>
<td>t_create</td>
</tr>
<tr>
<td></td>
<td>t_change</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

### instances

<table>
<thead>
<tr>
<th>Instance Id</th>
<th>Id1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
<td>TaskEntity</td>
</tr>
<tr>
<td><strong>Attributes Values</strong></td>
<td>taskstate: created</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instance Id</th>
<th>Id2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
<td>TaskEntity</td>
</tr>
<tr>
<td><strong>Attributes Values</strong></td>
<td>taskstate: active</td>
</tr>
</tbody>
</table>

---

**Fig. 4.** Relationship among meta-types, types and instances
When a type-level procedure is invoked, Called is provided as input parameter instantiated to a type identifier, whereas for instance-level procedures, Called is instantiated to an instance identifier.

For accessing instance-level attributes, the normal DML predicates are used as procedure parameters. Concerning type-level attributes, we can either read or update their value using call_proc (Caller, Called, read / write, Attr-Name, Attr-Value).

The implementation of the call_proc takes into consideration the inheritance mechanisms related to procedures and attributes.

5.3 Execution Manager and Planner

The EPOS Execution Manager (EM) and Planner work in close cooperation. The EM uses the “active” task type-properties (PRE-CODE-POST) of tasks to maintain a ready queue of runnable tasks, including the before-mentioned hotlist of these. The EM is responsible for invoking the Planner and for executing and re-executing the plans generated by this. Such plans are called task networks and are stored in the database for later (re)use.

The Planner works on a knowledge base consisting of type-descriptors and a Product Structure as its World State Description. It offers product-level assistance, such as construction of a derivation graph, and project-level assistance about job decomposition, and work agendas. The generated plan is handed over to the EM for later execution. In addition, the Planner offers re-plan due to Product Structure changes and/or execution failures.

Planner reasoning is based on the static PRE- and POST-conditions of task types. It applies hierarchical and non-linear planning. Hierarchical planning is accomplished by coupling the EM and the Planner: when the EM meets a task with empty CODE, it calls the Planner to decompose this task according to its DECOMPOSITION attribute. The Planner will take the current world state as the initial world state, the POST-condition of the parent task as the goal, and the tasks set in the parent’s DECOMPOSITION as the candidate subtask pool. It will then build a subplan to achieve the POST-condition, and add it to the original plan through the parent’s SubTasks relationship. The generated plan will be a partially ordered task network, to enable parallel processing and handle possible goal interactions.

The Planner can also be invoked from a normal procedure. An example is a trigger on a specialized Upd.E procedure, to automatically call a type-specific Compiler procedure after the update. This Compiler procedure may have to invoke the Planner to instantiate a separate compiler task, since an explicit derivation graph with networked tasks is needed for history recording. Then, this compiler task can be alerted by the trigger on the global Upd.E procedure (section 3.3).
5.4 The PM Manager

It consists of a batch SPELL Translator and interactive SPELL Editor (and Browser), and facilities to manage a type library and its evolution. Thus, the PM Manager is a meta-process tool.

The type-level procedures \texttt{t\_create}, \texttt{t\_delete}, and \texttt{t\_change} are defined both in the \texttt{TaskEntity} and \texttt{DataEntity} types and they may be redefined for their subtypes. Procedure \texttt{t\_create} creates a new subtype, by redefining type level attributes and adding/redefining procedures. It can be used to instrument the SPELL Translator/Editor with type-specific processing of certain type-level attributes, e.g. those for task types.

Obsolete types may be removed by \texttt{t\_delete}, unless referred to by other types or by remaining instances.

The \texttt{t\_change} procedure updates a type definition, and distinguishes between \texttt{hard} and \texttt{soft} type changes. The hard ones imply changes in instance-level attributes or in the subtype structure, and are generally disallowed. However, there is a possibility to modify the type of existing instances by \texttt{i\_convert}. The soft changes are the PM-specific ones, and \texttt{t\_change} will distinguish between a change in e.g. \texttt{CODE}, dynamic \texttt{PRE/POST-CONDITION}s, \texttt{FORMALS}, or procedures. Some delicate conversions and synchronizations may have to be performed.

The \texttt{i\_convert} rule attempts to convert an active task to a modified type definition, see the \texttt{update\_instance\_for\_redefined\_class} in CLOS [Kee89]. A task must be treated with regard to its state. The core of \texttt{i\_convert} does not depend on whether the type change has been done by subtyping or by versioning. In fact, redefinition of type-level properties in pre-defined types are not infrequent. Thus, \texttt{i\_convert} selectively considers the redefinition of such properties.

6 Conclusion and Future Work

SPELL is an extension of a former EPOS PML [COWL91], and object-oriented paradigms have provided a sound platform here. We can mention subtype extensibility, procedures as part of types, dynamic binding, declarable type-properties, explicit (meta-)types for model customization/versioning and extensibility (reflection).

But, maybe not all the SPELL functionality is needed, e.g. what is the practical distinction between tasks and triggers? Further, do we need a special EPOS PML, instead of using an existing object-oriented language such as CLOS, Smalltalk, or C++? We have rejected these latter languages, because they support almost no reasoning, no persistent data, only implicit relationships, limited tasking, and (partly) inadequate type-level modeling.

Prolog is suitable for database queries and for reasoning needed in our Planner. It provides flexible interpretation, and the chosen SWI-Prolog has a good
C interface. Also, the available Concurrent Prologs, did not fit for various reasons. A side-effect of EPOS PM has therefore been a fully object-oriented, quasi-concurrent, reflective and versioned Prolog! Our Prolog extensions are a small piece of code anyway, about 5,000 Prolog lines.

An exciting future challenge is to formalize the meta-processes of PM. We also need to develop better methods for PM design (by a high-level PML?), to clarify the methodological issues in versioning versus subtyping, to support model elicitation and assessment, and to structure and evolve large (reusable) process models.

small

References


