A Reference Model for Open Source Projects

Michel Pawlak and Ciarán Bryce

Abstract. Free and Open Source Software (F/OSS) constitutes a large class of the software used today. Such software is produced by a self-organizing community that has no centralized control. The absence of control can lead to operational and efficiency problems for large-sized projects. This paper proposes a process reference model (PRM) and model that captures the activities, roles and resources of the process, that allows reasoning about process coherency and efficiency.

1 Introduction

Free and Open Source Software (F/OSS) is one of the great facts of software development of the past few years. In this model, a community of people with common interests collaborate to produce software. The software is distributed with the source code which can be freely modified by any developer. The community is self-organizing – as opposed to a large software company, there is no hierarchal control. F/OSS is said to be organized like a *bazaar* as opposed to the *cathedral* model of proprietary software development used by companies [15]. In a F/OSS project, the interests of the community push design requirements and software licenses regulate IPR. A F/OSS community is responsible for all aspects of software development, from requirements to coding, testing, and even manual compilation and translation. Major examples of F/OSS projects are Linux, Apache, Eclipse and OpenOffice.

Increasing the size of a F/OSS project community brings great potential to the project – more ideas, code and developers. However, since there is no hierarchal control or regulated coordination, projects can suffer from inefficiency and operational concerns as they become large. For instance, there might be several developers working on a patch for the same code package, unaware of the efforts of each other. Further, it can be hard to locate a community member who may be helpful for a certain task (e.g., to fix or develop a specialized package, organize a seminar on the project's software). A F/OSS project needs mechanisms for improved information availability, not just to address efficiency concerns, but also for correctness with respect to project artifacts. For instance, each package metadata must include complete dependency information and a complete list of patches; it should not be possible to create artifacts that do not have the complete set of meta-data.

F/OSS is an example of a virtual process – a set of activities (e.g., coding, testing, community management, etc. in F/OSS) exploiting resources (e.g., packages, configurations) and roles (e.g., developers, project committers). This paper proposes a model – denoted the *Process Reference Model* (PRM) – that formalizes the F/OSS process. The model permits meta-data, or *attributes*, to be bound to artifacts to simplify the classification and localization of artifacts. Attributes express information such as defect reports, patches, configuration requirements, topics. Attributes facilitate coordination between activities since all information required by an activity (e.g., patch development) that is produced by another activity (e.g., testing) is expressed in the artifact attributes (e.g., defect report). The process model can also permit different F/OSS projects to be compared and contrasted based on the activities each undertake. Existing project methodologies and supporting tools [10, 18] tend to be limited to project artifacts, ignore the organizational aspects of the process or ignore process interactions. We contend that F/OSS processes can only be improved by addressing process issues.

The remainder of this paper is organized as follows. Section 2 presents the F/OSS model, and highlight inefficiencies that can occur as projects grow in size. Section 3 presents an overview of the reference model, and Section 4 illustrates how the model can be used. Related work is presented in Section 5 and Section 6 concludes the paper.

2 F/OSS Process Management

As users of open source projects know, several technical problems appear as the project grows in size. One is dependency management (the problem of identifying and locating the set of packages that need to be installed – or removed – when a given package is installed). This problem increases as the frequency of releases increases. Another operational problem is testing: many F/OSS software errors are configuration errors that cannot be detected by the distributor given the multitude of configurations that he needs to test. These errors are only detected once the software is deployed on the clients. A third operational problem is code distribution, and in particular, the overhead of distributing software to a huge number of end-clients. As the number of users grow, then the latency involved in downloading software from one of a set of mirror servers increases, especially when releases are frequent, as does the effort needed to keep the mirrors up-to-date. Errors or latency problems during code distribution can lead to inconsistencies in the software installed on an end-user machine.

F/OSS is more than the simple production, testing and deployment of software. It also involves activities such as community management, organization of seminars, production of manuals, etc. A community member may even decide to start a new activity related to the project, and this can be as varied as starting a sub-project, fund-raising or server maintenance. The plethora of activities poses the following organizational requirements:

- Possibility of locating competence in the community. Managing an activity entails harnessing the competence of community members. Competent and potentially interested members must be located. Apart from news-groups and mailing lists, there is no way of actively locating potential activity collaborators.

- Information about activities and participants needs to be made available to the community. For instance, a user seeking to install a package must be immediately informed of any detected configuration error. Similarly, coding activities must be aware of information from testing; development activities need to be informed of information on community profiles produced by community management activities.

The technical issues mentioned above are also related to inadequate information flow between activities. In the case of F/OSS dependency management, there is inefficient information flow between the development of packages activity and the download and installation activities. In the case of configuration defects, clearly there is a need for a testing and defect reporting activity that is closely intertwined with the download activity. To optimize downloading, it is important for a client to precisely specify the packages or type of software he requires so that superfluous packages are not transferred.

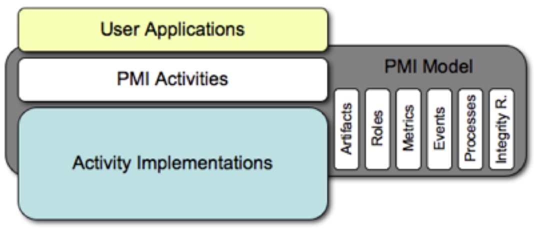

Fig. 1. PRM Overview

To address these issues, we have developed a process model for F/OSS that encapsulates all activities and artifacts.

- The core data types of the model represent each F/OSS *artifact*; a type is defined for each entity managed within a project, e.g., packages, test reports, community members, projects, etc. An artifact definition specifies *attribute* artifacts to capture the relationship between artifacts, e.g., a package artifacts must have a valid software license as an attribute.
- The description layer of the model defines artifacts related to management. An *activity* for instance formalizes a set of operations with a well-defined purpose, e.g., testing operations, community management operations, documentation translation procedures, etc. A *process* is a set of computational steps, based on activity operations. It encapsulates the actions that some member of a F/OSS project is responsible for. A process is a first-class entity in our model – it can be declared and instantiated. An *Actor* is any member of the F/OSS project community. Actors execute processes, but for security and management reasons, the actors permitted to execute each process is controlled. Thus, each actor is assigned a set of *Roles* that determine his *Rights* to artifacts in the scope of the project.

- The instantiation of a process is known as a *task*. A running task has an actor and role associated. *Events* are asynchronous communication objects for processes. For instance, a bug fixing process that specializes in kernel bugs is programmed to react to events representing kernel defect reports; these events would be generated by a kernel test process.
- The final layer of the model concerns measurement. One of the core requirements of any process is performance feedback. These can be varied in nature; in F/OSS, examples can include the number of bugs per month, the number of developers specialized in cryptography, the number of code submissions, etc. The model thus permits *metrics* to be dynamically declared and bound to artifacts, and a *log* to be maintained.

We contend that a model that adopts this approach is the basis for addressing the efficiency concerns of the F/OSS process:

1. *Flexibility* of look-up. An end-user is able to locate code units based on properties such as functionality, platform requirements or license. Similarly, a community member can locate other members by specifying competences or topics of interest.
2. *Integrity* of artifacts. The model not only defines F/OSS attributes, but operations to create and manipulate artifacts. These operations ensure that the precise set of attributes are bound to an artifact. Thus, a code package cannot exist without the exact set of dependency information and a license attribute.
3. *Cross-activity coordination*. Attributes clarifies interaction between activities. For instance, defect reporting entails binding defect report attributes to code artifacts; these in turn can be localized by participants of a debugging activity based on the report attributes.

3 PRM Elements

The core of the PRM is a set of data types that define the main F/OSS *artifacts*. F/OSS processes are composed of *activities* that manipulate artifacts; these capture the execution aspect of processes. We look at each in turn, and then give an overview of how the PRM can be used.

3.1 Artifacts and Attributes

As mentioned, an Artifact is any F/OSS resource. Artifacts are typed; the type defines the operations applicable as well as possible constraints on the artifact during its lifecycle. The set of artifacts defined by the PRM are listed in Table 1.

Some artifacts may serve as attributes to other artifacts; for instance, each project artifact has a list of topic artifacts as attributes (to describe the domain of the project). Attributes can be combined in Boolean expressions and *directory* expressions. Expressions are useful for locating artifacts based on a combination of different attributes, using the PRM's lookup primitive. For instance, when searching for content, one can specify that it comes from any mirror server (s_1, s_2, ..., s_m) but not from

Artifact	Description
Actor	F/OSS Community member taking part in diverse activities of the F/OSS Process
Activity	Operations related to a specific domain, such as Testing, Production, Community Management
Date	Simple date using any format
Event	Artifact used for asynchronous communication and synchronization
EventType	Type of the Event
IntegrityRule	Rule to be enforced for ensuring the integrity of Process execution
Log	Capture of Actor behavior
Metric	Reusable user-defined measure of any aspect of the F/OSS Process made in order to improve it
MetricOccurence	Information concerning when a measure has to be taken
MetricSet	Set defining a logical relation between different measurements
MetricViewPoint	General context in which a measure is done
Process	Orchestration of F/OSS operations, possibly taking up time, expertise or other resource.
Project	Umbrella indicating F/OSS Processes, Actors and Roles involved in the life cycle of a set of
ProcessCategory	The category the Process belongs to
ProcessType	Type of the Process, can be *once-off* or *recurring*
Right	Allowed Action in the scope of a Task
Role	Responsibility within the F/OSS process expressed as a collection of Processes
Task	Assignment of a Process to an Actor in the scope of a Project and Role
TaskStatus	Current Status of the Task
Text	Plain text information
Topic	Human readable information that may be used to express interests, knowledge, competences

Table 1. F/OSS Artifacts Overview.

an unofficial mirror server ($s_{m+1}, s_{m+2}, ..., s_n$); this is expressed using the Boolean expression $(s_1 \vee s_2 ..... \vee s_m) \wedge \overline{(s_{m+1} \vee s_{m+2} ..... \vee s_n)}$.

A directory expression combines artifacts of different types. The look-up can take all forms of Artifact as argument. We use the term *directory* to denote a grouping of Artifacts of different types. $\mathcal{A} \sqsubset \mathcal{D}$ is an expression that evaluates to true if, and only if, the directory $\mathcal{D}$ contains the Artifact set $\mathcal{A}$. Directories are built from Artifact sets and Directories using the $+$ and $-$ operators. The semantics of directories is defined in Table 2. A particular Artifact set $\mathcal{A}_i$ contained in a directory $\mathcal{D}$ is selected using the (.) operator. The $\ll_i$ operator is a utility function that defines a directory from an existing directory; the two directories differ by a component set.

$\mathcal{A}_i + \mathcal{A}_j = \mathcal{D}$	$\Rightarrow \mathcal{A}_i \sqsubset \mathcal{D} \wedge \mathcal{A}_j \sqsubset \mathcal{D}$
$\mathcal{D}_1 + \mathcal{A}_j = \mathcal{D}_2$	$\Rightarrow \mathcal{A}_j \sqsubset \mathcal{D}_2 \wedge \forall \mathcal{A}_i \sqsubset \mathcal{D}_1, \mathcal{A}_i \sqsubset \mathcal{D}_2$
$\mathcal{D}_1 + \mathcal{D}_2 = \mathcal{D}_3$	$\Rightarrow \forall \mathcal{A}_i \sqsubset \mathcal{D}_1, \mathcal{A}_i \sqsubset \mathcal{D}_3 \wedge \forall \mathcal{A}_j \sqsubset \mathcal{D}_2, \mathcal{A}_j \sqsubset \mathcal{D}_3$
$\mathcal{D}_{ij} - \mathcal{A}_j = \mathcal{D}_i$	$\Rightarrow \neg (\mathcal{A}_j \sqsubset \mathcal{D}_i) \wedge \forall \mathcal{A}_i \sqsubset \mathcal{D}_{ij}, \mathcal{A}_i \sqsubset \mathcal{D}_i \wedge \mathcal{A}_i \neq \mathcal{A}_j$
$\mathcal{D}_{ij} - \mathcal{D}_j = \mathcal{D}_i$	$\Rightarrow \forall \mathcal{A}_i \sqsubset \mathcal{D}_{ij}, \mathcal{A}_i \sqsubset \mathcal{D}_i \wedge \forall \mathcal{A}_j \sqsubset \mathcal{D}_j, \neg (\mathcal{A}_j \sqsubset \mathcal{D}_i) \wedge \mathcal{A}_i \neq \mathcal{A}_j$
$\mathcal{D}.i$	$= \mathcal{A}_i$
$\mathcal{D} \ll_i \mathcal{A}_i$'	$= \mathcal{D} - \mathcal{D}.i + \mathcal{A}_i$'

Table 2. Artifact set and directory algebra.

The lookup operation is a matching process in which the a directory of attributes is specified. Each Artifact type component is compared through the matching primitive *m()* with the corresponding target type; where $m(a_{template}, a_{target})$ verifies if $a_{template}$ is substitutable with a_{target}. That is,

$$m(a_1, a_2) \quad \Leftrightarrow \quad a_1 \simeq a_2$$

The matching semantics for artifact expressions are defined in Tables 3 and 4.

$m(A_{template}, A_{target})$	$\Leftrightarrow \forall a_j \in A_{target} . \exists a_i \in A_{template} . m(a_i, a_j)$
$m(D_{template}, D_{target})$	$\Leftrightarrow \forall D_{template}.i \;.\; \exists D_{target}.j \;.\; m(D_{template}.i, D_{target}.j) \wedge i = j$

Table 3. Semantics of Attribute Set and Directory matching.

$m(e_{template1} \wedge e_{template2}, A_{target})$	$\Leftrightarrow m(e_{template1}, A_{target}) \wedge m(e_{template2}, A_{target})$
$m(e_{template1} \vee e_{template2}, A_{target})$	$\Leftrightarrow m(e_{template1}, A_{target}) \vee m(e_{template2}, A_{target})$
$m(\overline{e_{template}}, A_{target})$	$\Leftrightarrow \neg\, m(e_{template}, A_{target})$

Table 4. Semantics of expression matching.

3.2 PRM Primitives

The PRM defines a core set of operations that manipulate artifacts; these are listed in Tables 5 and 6. The core PRM operations are the bricks from which F/OSS processes are specified.

The PRM primitive operations defined for artifacts allow new artifact types to be defined. Each artifact definition requires a definition of substitutability for the matching process and a set of integrity rules for associated attributes.

An *activity* is a set of user-defined operations that relate to some aspect of F/OSS. Common F/OSS activities include Community Management, Metrics Management, Rights Management, Projects Management, Distribution Management, Production Management and Defects Management. The PRM primitives allow for the definition of an activity and its binding to a project. Each activity has a set of user-defined operations (denoted λ) that are themselves either defined using the core PRM primitives, or are considered as black-box operations whose implementation is opaque to the PRM. Some examples of this are given in the next sub-section.

Context	Name	Description
Artifact	registerArtifactType($\mathbb{P}\,\mathcal{IR}$, Project):Type	Registers a new Artifact Type for a Project
	newArtifact(Type, Values Directory, Project):Artifact	Instantiates a new Artifact for a Project
	lookup(D,$\mathbb{P}$Artifact):$\mathbb{P}$Artifact	Attribute aware Artifact search
	exists(Artifact):Boolean	verifies if an Artifact exists
	substitutable(Artifact, Artifact):Boolean	Substitutability verification
Activity	declareActivity(Name, $\mathbb{P}$(λ, $\mathbb{P}\,\mathcal{IR}$)):Activity	New Activity declaration in terms of operations
	associateActivity(Activity, Project):void	Associates an Activity to a Project
	lookup(D):$\mathbb{P}$Artifact	Attribute aware Activity search
	exists(Activity):Boolean	verifies if an Activity exists
	isAssociated(Activity, Project):Boolean	verifies if an Activity exists
Actor	registerActor(D):Actor	Registers a new Actor
	setContactInfo(D):void	Defines Actor's Contact information
	setCompetence(D):void	Defines a Competence
	setInterest(D):void	Defines an Interest
	setKnowledge(D):void	Defines a Knowledge
	lookup(D):$\mathbb{P}$Artifact	Attribute aware Actor search
	exists(Actor):Boolean	verifies if an Actor exists
Event	raise(Event):void	Event raise
	registerListener(EventListener):void	Registers an EventListener
	unregisterListener(EventListener):void	Unregisters an EventListener
	observe(Event, Time):Event	Synchronization on an Event, waiting until it is raised
	exists(Event):Boolean	verifies if an Event exists
EventListener	raised(Event):void	Notifies the Listener that an Event has been raised
Log	createLog(Activity, λ, $D_{parameters}$, D_{result}):Log	Logs a call to a PRM operation
	lookup(D):$\mathbb{P}$Artifact	Attribute aware Log search
	query(Text$_{queryexpression}$, D$_{queryparameters}$):D	Log querying
	exists(Log):Boolean	verifies if a Log exists
Metric	newMetric(Name, OccurExpr, MeasExpr):Metric	New Metric creation
	registerMetric(Metric):void	Metric registration
	enableMetric(Metric, Project):void	Enables Metric usage in the scope of a Project
	enabledMetric(Metric, Project):void	Checks Metric availability in the scope of a Project
	disableMetric(Metric, Project):void	Disables Metric usage in the scope of a Project
	evaluateMetric(Metric, Project, Actor):Number	Metric execution by an Actor in the scope of a Project
	usesMetrics(Metric, Metric):Boolean	Checks if a Metric depends on another Metric
	dependsOnMetrics(Metric):$\mathbb{P}$Metric	Metrics used by a Metric
	usedByMetrics(Metric):$\mathbb{P}$Metric	Metrics using a Metric
	lookup(D):$\mathbb{P}$Artifact	Attribute aware Metric search
	exists(Metric):Boolean	verifies if a Metric exists
Process	declareProcess(Name, ProcessStep$_{first}$, $D_{interests}$, $D_{knowledge}$, $D_{competences}$, ProcessType):Process	New Process creation for a Project
	declareProcessStep(Activity, λ):Step	Declares a Step for a Process as a call to an operation
	declareConditionStep(Step$_{previous}$, Condition, Step$_{iftrue}$, Step$_{iffalse}$):Step	Step sequencing on condition
	declareLoopStep(Step$_{previous}$, Condition, Step$_{whiletrue}$, Step$_{whenfalse}$): ProcessStep	Step sequencing on defined condition
	setProcessStepSequence(Step$_{previous}$, Step$_{next}$):void	Step sequencing
	lookup(D):$\mathbb{P}$Artifact	Attribute aware Process search
	exists(Process):Boolean	verifies if a Process exists
	exists(ProcessStep):Boolean	verifies if a ProcessStep exists

Table 5. PRM operations (part 1).

Context	Name	Description
Project	declareProject(Text$_{Name}$, Text$_{Acronym}$, Text$_{Description}$, Actor, $\mathbb{P}$Topic, Url, Project$_{Parent}$):Project	New Project creation
	setTopics($\mathbb{P}$Topic):void	sets the Topics the project is working on
	setDescription(Text):void	sets the description of the Project
	setContact(Actor):void	sets the contact Actor for the Project
	registerContributor(Actor, Project):void	registers an Actor as a Project contributor
	getContributors(Project):$\mathbb{P}$Actor	returns the contributors for a Project
	getProjects(Actor):$\mathbb{P}$Project	returns a list of Project an Actor contributes to
	lookup(D):$\mathbb{P}$Artifact	Attribute aware Project search
	exists(Project):Boolean	verifies if a Project exists
Right	declareRight(Activity, λ, Task):Right	New Right declaration in the scope of a project and Task
	extendRight(Right, Right):Right	Right extension with another Right
	assignRight(Right, Actor):void	Right assignment to an Actor
	removeRight(Right, Actor):void	Right removal to an Actor
	allowed(Actor, Project, Process, Activity, λ):boolean	Activity Operation access verification
	lookup(D):$\mathbb{P}$Artifact	Attribute aware Right search
	exists(Right):Boolean	verifies if a Right exists
Role	declareRole(Text$_n$*ame*, Text$_d$*escription*, $\mathbb{P}$Process)	New Role declaration
	associateRole(Role, Project):void	Associates a Role to a Project
	acceptRole(Role, Actor, Project):void	Proposes a Role to an Actor
	assignRole(Role, Actor, Project):void	Associates a Role to an Actor
	containsProcess(Role, Process):Boolean	Verifies if a Process is part of a Role
	isRoleAssociated(Role, Project):Boolean	Verifies if Role is associated to a Project
	isRoleAssigned(Role, Actor, Project):Boolean	Verifies if role assigned to an Actor in the scope of Project
	isRoleAssigned(Role, Project):Boolean	Verifies if role assigned to any Actor in scope of Project
	getRoles(Project):$\mathbb{P}$Role	Returns the Roles associated to a Project
	getUnassignedRoles(Project):$\mathbb{P}$Role	Returns roles associated with project, not assigned to actor
	getRoles(Actor, Project):$\mathbb{P}$Role	Returns roles assigned to actor for a Project
	getContributors(Role, Project):$\mathbb{P}$Actor	returns the contributors of a Project having a Role
	lookup(D):$\mathbb{P}$Artifact	Attribute aware Role search
	exists(Role):Boolean	verifies if a Role exists
	proposeRole(Role, Actor, Project):void	Proposes a Role to an Actor
	proposeRoleContribution(Role, Actor, Project):void	Allows Actors to proposes their participation for a Role
	answerRoleProposition(Role, Actor, Project, Boolean)	Allows Actors to answer a role proposition
	isRoleProposed(Role, Actor, Project):Boolean	Verifies if role proposed to an Actor in the scope of a Project
	isRoleAccepted(Role, Actor, Project):Boolean	Verifies if role accepted by an Actor in the scope of a Project
	getRolesProposed(Actor, Project):$\mathbb{P}$Role	Returns the Roles proposed to an Actor for a Project
	getContributorsPropositions(Role, Project):$\mathbb{P}$Actor	Returns Actors proposed contribution for role in Project
	getPotentialContributors(Role, Project):$\mathbb{P}$Actor	returns the contributors of a Project having a Role
Task	newTask(Actor, Project, Process, Role)	New Task creation
	getTasks(Actor, Project):$\mathbb{P}$Task	Returns the Tasks an Actor has to do for a Project
	getTasks(Actor, Project, Role):$\mathbb{P}$Task	Returns the Tasks an Actor has to do for a Role in a Project
	changeTaskStatus(Task, TaskStatus):void	Changes the status of the Task
	executeTask(Task, D):D	Executes first Step of the Task providing a Directory of parameters and returns the result
	executeStep(Task, ProcessStep, D):D	Executes a ProcessStep providing a Directory of parameters
	nextStep(Task):ProcessStep	Indicates the next Step to be executed.
	currentStepOfTask(Task):ProcessStep	Returns current position in Process
	lookup(D):$\mathbb{P}$Artifact	Attribute aware Task search
	exists(Task):Boolean	verifies if a Task exists

Table 6. PRM operations (part 2).

The *Actor* PRM Artifact represents any contributor to the F/OSS Process. It can be a person or an institution. Each actor has topic attributes that represent his interests, knowledge and competence. The primitives take a directory expression D as argument that combine these elements.

The PRM allows for the publication and lookup of F/OSS information by activities. There is thus a need for coordination, where an activity can sleep waiting for an artifact to be published, and be automatically informed of the publication. The means for coordination in the PRM are furnished by *events*. There are two main types of events: i) Execution Events are raised when a PRM operation is called; this event type holds context information such as the operation having been executed, the Task which triggered it as well as temporal information. ii) Alarm Events are raised to indicate that a special state is reached. Like for Execution Events, they carry context information. The observe primitive is a blocking operation that waits for an artifact with the specified attribute directory to be published.

Metrics are registered with the PRM. They can be calculated on-demand using the evaluate primitive. However, measures can also be taken each time a given PRM Activity operation $a.\,\lambda$ is called in order to handle dynamic metrics that have to be continuously measured. The core of the PRM metrics management is described in Figure 2. The operation $A.\,\lambda$ must return a number Artifact or if it not the case, the result must be treated by a numerize mechanism that returns a number.

$$
\begin{array}{lcl}
\textit{measure} & ::= & a.\,\lambda(D_{param}) \rightarrow \textit{Number} \\
 & | & \textit{numerize}(\textit{measure}) \rightarrow \textit{Number} \\
 & & \\
\textit{metricExpression} & ::= & \textit{operand } s.t.\ \textit{operand} \in \{\mathbb{R}, \textit{measure}\} \\
 & | & \textit{operand } (\textit{op metricExpression})_{cond}\ s.t.\ \textit{op} \in \{+, -, *, /, \%\}
\end{array}
$$

Fig. 2. Metric Execution Expression definition

Process execution is regulated by *roles*. Typical roles in a F/OSS project include committer, project manger, tester, etc. Roles can be proposed and assigned to actors. Access control for security is implemented using *rights*. The PRM primitives allow rights to be defined for activity operations and these can be assigned to actors.

A *process* artifact is a programmatic expression of a set of activity operations. The structure of a process π is given in Figure 3. Finally, a *task* is an instantiation of a process with specific actors and roles.

3.3 Example: Process Handling

This section presents a short example whose aim is to give a favor of the PRM. The example concerns a small F/OSS community that handles university courses. The artifacts produced by the community are slides and exams. The activities of the

$$\begin{array}{rcl} \pi_i & ::= & \pi_j;\ \pi_k \\ & | & \pi_j \mid\mid \pi_k \\ & | & cond\pi_j\pi_k \\ & | & *cond\pi_j \\ & | & A.\,\lambda(D) \rightarrow v \\ & | & nil \end{array}$$

Fig. 3. Process Structure

community centre around teaching, taking exams and continuous assessment. The advantage of organizing a course around the open source model is that the community can contribute knowledge to the preparation of classes on specialized topics.

The first step is to define the initial set of artifacts and actors. The initial set of actors have competence in programming, operating systems and security. The courses are composed of slide presentations and exams. The set of topics of a course are the sum of the topics of each individual presentation and exam (example integrity rule).

```
// Define topics
security = newArtifact(TOPIC, "Security");
programming = newArtifact(TOPIC, "Programming");
os = newArtifact(TOPIC, "OS);

// Actors
ciaran = registerActor("Ciaran"...);
ciaran.setInterests(security and os);
michel = registerActor("Michel"...);
michel.setInterests(programming);

// Course are the raw material produced. Use slides and examinations
//
registerArtifactType("Slides", attr = { topics : Set.TOPIC; creator : ACTOR} );
registerArtifactType("Exams", attr = { topics : Set.TOPIC; creator : ACTOR} );

registerArtifactType("Course", attr = { topics : Set.TOPIC; creator : ACTOR;
                      allSlides : Set.Slides; allExams : Set.Exams}
           IR = { topics == union allExams.topics + all Slides.topics } );
```

The next stage in the community management example is to define the activities. One of the key activities is Teaching. The operations associated with this activity are ProduceSlides, GiveLecture, SetExam, ControlExam. The first of these operations is defined as

```
ProduceSlides(name: Text, T : Set.TOPICS, A : ACTOR, C : COURSE) =
        { newArtifact(Slides, name, T + A + C); }
```

where the attributes specified in as argument to the operation become the attributes of the new slides artifact. The second operation, GiveLecture, is opaque in that it is not defined in terms of the PRM's primitive operations. A process for a course, e.g., DistributedSystems can now be declared:

```
Process DistributedSystems(creator : ACTOR) =
let intro = ProduceSlides("Introduction", {os}, creator)
| let network = ProduceSlides("Network", {os, security}, creator)
| let filesys = ProduceSlides("FileSystems", {os, security}, creator)
| let p2p =  ProduceSlides("P2P", {os, security}, creator);
GiveLecture(intro); GiveLecture(network);
GiveLecture(filesys); GiveLecture(p2p);
let ex1 = SetExam(intro, ...) | let ex2 = SetExam(network, ...)
| let ex3 = SetExam(filesys, ...) | let ex4 = SetExam(p2p, ...);
ControlExam(ex1); ControlExam(ex2);
ControlExam(ex3); ControlExam(ex4);
```

Now that the process has been defined, the final step is to permit its execution as a task. This entails assigning access rights for the process activities to actors, and then instantiating the task. An access right is defined for each of the operations of each activity:

```
let r1 = declareRight(Teaching, ProduceSlides);
```

Then, a task can be specified that binds rights, roles and actors:

```
task t = newTask(michel, DistributedSystems, Coordinator);
```

So long as the actor Michel has the necessary access rights, then he can coordinate a course on distributed systems.

The full power of the model can now be exploited. For instance, actors who are interested in giving or taking courses can register with the system. They can locate courses based on the topics of interest. Actors who wish to coordinate their own courses can locate slides and exams in the community based on their keyword search. Potential students can also locate courses of interest to them by defining processes that look-up courses. Further, actors can define new roles and activities for issues such as course translation, course classification, or any such extensions.

4 Related Work

As [16] highlights, the motivation of contributors to F/OSS projects is central to project success. The author argues that the motivation is not just related to computer science interests, but also to economics, law, psychology and anthropology. The author recommends that all factors that motivate volunteers to contribute to a project be considered when reorganizing the structure. Changing the internal structure or policy in an existing project, providing a new approach to the F/OSS Process can therefore have a strong impact on the success of projects and as such managing community resources must be as important as managing content resources to ensure F/OSS projects success. This is an essential aim of the PRM.

An overview of how F/OSS projects are organized is presented in [7]. The paper explains related terminology and overviews main involved processes. The processes include decision-making within the project management, accountability of bugs to packages, communication among developers, generation of awareness about the project in the software community, managing source code, testing and release management. The PRM aims at providing a means to interact with theses processes and link them to a model and F/OSS resource description in order to have a global view.

A large number of existing tools and solutions aim at tackling issues related to specific activities within the F/OSS process. These solutions cover, for instance, defect management (Bugzilla [3]), testing (Bugzilla test runner [17], Fitnesse [8], Salome [19]), projects provide dashboards [5, 4], and some other tools (SourceForge) [18], GForge [11], Alioth [2], Libresource [13]) provide integrated solutions for managing F/OSS projects covering different aspects. Nevertheless, none provide a transversal and generic approach to F/OSS, none consider the F/OSS Process as a whole highlighting how activities are interwoven and they do not manage communities.

The Flink group [9] aims at demonstrating the benefits that will result from formalizing knowledge about the Linux operating system. In [12] authors discusses the application of ontology-based knowledge engineering to the domain of Linux. Various applications such as package management or information search are discussed which would all benefit from a comprehensive ontology of the domain. The use of an ontology for describing Linux is similar to our approach as it aims at providing a common ground of understanding. PRM Artifacts can be thus considered as an ontology. Nevertheless, the PRM relies on a model providing advanced features such as substitutability, look-up, matching or directories which allow to define integrity rules.

QSOS is a method for qualifying, comparing and selecting F/OSS in an objective, traceable and argued way [10]. It relies on interdependent and iterative steps aimed at generating software ID cards and evaluation sheets which can be used in order to support the selection of the best solution in a given context. Though less multi-purpose than the PRM, it would be interesting to investigate how the QSOS method could be built on top of the PRM.

The OpenSuse *build service* [14] provides a complete distribution development platform to create Linux distributions based on SUSE Linux. Its open interfaces allow external services to interact with the build service and use its resources. Like for the PRM, the OpenSuse build service aims at connecting open source communities, providing means to develop distributions while controlling issues like dependencies. Nevertheless, while the build service focuses at tackling the distribution development issue, this is only one aspect of the PRM.

Red Hat Network (RHN) is an architecture whose design is also articulated around an API [20]. As for OpenSuse, its scope is narrower than the PRM's, since it essentially focuses on code distribution. RHN is used to download distribution ISOs, patches and software packages as well as to update systems based on user customization. The network is accessible through an *Access API*. The key abstraction RHN provides is the notion of *channels*, which corresponds to a set of packages. Every client machine that is connected to a specific channel can be updated when the content of the channel changes. Access rights are associated with channels which allows to have control over the systems having access to it. Instead of channels, the PRM provides interfaces for different F/OSS processes and defines integrity rules to be enforced.

5 Conclusion

This paper presented a process reference model (PRM) to model the activities, roles and resources of the F/OSS process which allows F/OSS activities to be more efficiently handled. The goal of the PRM is to define the key content and community artifacts of the F/OSS process and to formalize the relations between these. We believe that this precision allows inefficiencies in F/OSS processes to be detected and eliminated.

The PRM can be used in several ways. For instance, it can serve as a basis for comparing the processes used by various F/OSS distributor. It can also serve as the design of an information system for a new project. This information system would act as a real-time dashboard for the project and ensure that the integrity rules are respected for all F/OSS operations.

Work on the PRM is continuing. One current step is the definition of further integrity rules for F/OSS operations. Another is the application of the PRM to use-case scenarios from F/OSS within the scope of the EU EDOS Project (Environment for the development and Distribution of Open Source software) [1, 6], a project aiming at developing technology and tools to support and improve the production and customization processes of F/OSS distributions.

Acknowledgments

This work is being conducted in the context of the EU 6th Framework project EDOS (grant number FP6-IST-004312).

References

1. Serge Abiteboul, Roberto Di Cosmo, Stefane Fermigier, Stephane Lauriere, and al. Edos: Environment for the development and distribution of open source software. In *Proceedings of the First International Conference on Open Source Systems, Genova, Italy*, July 2005.
2. Alioth. http://alioth.debian.org/, June 2006.
3. Bugzilla. http://www.mozilla.org/bugs/, June 2006.
4. FreeBSD Dashboard. http://people.freebsd.org/ bsd/prstats/, June 2006.
5. Eclipse Project Dashboards. http://www.eclipse.org/projects/dashboard/, June 2006.
6. EDOS. Project website. http://www.edos-project.org, June 2006.
7. J. Erenkrantz and R. Taylor. Supporting distributed and decentralized projects: Drawing lessons from the open source community. In *Proceedings of 1st Workshop on Open Source in an Industrial Context, Anaheim, California.*, October 2003.
8. Fitnesse. http://www.fitnesse.org, June 2006.
9. Formalized Linux Knowledge (Flink). http://flink.dcc.ufba.br/en/, June 2006.
10. Method for Qualification and Selection of Open Source software (QSOS). http://www.qsos.org/, june 2006.
11. GForge. http://gforge.org/, June 2006.
12. Debora Abdalla Guillaume Barreau. Semantic linux: a fertile ground for the semantic web, April 2005.

13. Libresource. http://dev.libresource.org/, June 2006.
14. OpenSuse. Opensuse build service, http://en.opensuse.org/build_service, June 2006.
15. Eric. S. Raymond. The cathedral and the bazaar. http://www.openresources.com/documents/cathedral-bazaar/, August 1998.
16. Maria Alessandra Rossi. Decoding the free/open source(f/oss) software puzzle a survey of theoretical and empirical contributions, April 2004.
17. Bugzilla Test Runner. http://www.willowriver.net/products/testrunner.php, June 2006.
18. Sourceforge. http://sourceforge.net/, June 2006.
19. Salome Test Management Tool. https://wiki.objectweb.org/salome-tmf/, June 2006.
20. Sean Witty. Best Practices for Deploying and Managing Linux with Red Hat Network, December 2004.