Annexe:
Using Features To Structure the F/OSS Distribution Process

Michel Pawlak, Ciarán Bryce, Michel Deriaz
University of Geneva

Draft: June 9, 2005

Background

The note *Towards an Edos API: Modelling the* F/OSS *Process* identifies some content types for F/OSS and the EDOS API, including ContentUnit, Bundle and InstallableUnit. Distinguishing types helps us to identify the information flows in the process and the user roles.

The next step is to consider the attributes or *features* of these types. This allows us to define and formalise dependency relations between type instances such as patch_for or upgrade_of. It also allows instances to be precisely designated so that an end-user can locate the installable unit that best matches his requirements, and a developer can locate the content units (e.g., libraries) that best suits the requirements of his application.

Key point. Features describe units of content, which is required to be able to locate and distribute units. In current F/OSS, the features are limited to version numbers and filenames. Licenses can be considered a feature but no distribution system allows a search for units of content based on their licenses. By extending the notion of content features to concepts like application-specific ontologies, licenses, signatures, etc., a great deal of flexibility for the distribution architecture can be obtained.

Recap Of Key Content Types

The following types were defined in *Towards an Edos API: Modelling the* F/OSS *Process*.

- A Project produces and distributes F/OSS software content to the community. The content is generally created by a group of developers who submit their software directly to project committers, or indirectly via public domain software sites like *freshmeat*. A Project may depend on other projects, by including distributed software in its own distributed software.
- A ContentUnit is the smallest unit of production in the project. The nature of the content is not important – it can be source code, binary code or documentation. It is not distributed by a project but rather is incorporated into an application. Thus, it is distributed along with the other ContentUnits of the same application.

- A Bundle is the project's internal view of a F/OSS content that is offered to the community. As an application may consist of several ContentUnits, e.g., for libraries, binary or source code and documentation, a Bundles is the larger unit of production that captures this composition. Once composed, the Bundle is attributed a License that covers all units of the bundle. A Bundle is a recursive structure since different components of the application may be covered by different licenses, for instance, a project specific license and an LGPL license. In this case, there is a bundle for each different license, where each bundle contains the distribution units covered by that license, and the related bundles are packed into a envelope bundle with a project specific bundle license compatible with both licences.

- An InstallableUnit is a representation of a Bundle that can be installed on an end-user machine. Using an object-oriented analogy, an InstallableUnit is an instance of a Bundle. Thus an InstallableUnit is what the end-user sees and corresponds to his environment, e.g., it contains the applications that the end-user in question wants to install or the object code with all needed information for the unit to be compiled for his target architecture. A project may produce and make available a number of InstallableUnits F/OSS for a given Bundle.

Features

It is time to consider the nature of these types in more detail, in particular, their attributes or *features*. These are needed to understand and formalise the F/OSS relations between types. Each of the types above has features, though here we concentrate on ContentUnit, Bundle and InstallableUnit.

A feature is any meta information that can be meaningfully associated with a content type or its instance. A feature can be statically associated with a content type, e.g., date of creation, author of unit, function signatures; alternatively, the features may be associated with instances of types, e.g., performance or popularity. Features may also represent an ontology specific to a project or domain (e.g., testing); such ontologies may be defined during the lifetime of the project. Finally, the scope of features can vary; some are internal to the project – such as coding conventions – while others are specific to certain roles, e.g., stability.

Features allow units of content to be identified, which benefits both end-users and developers.

- It allows the end-user to locate software by category. For instance, a user should be able to search for *a text processor able to include some multimedia files like pictures*. In existing F/OSS projects, the feature abstraction is artificially added through categories and keywords on top of a distribution model mainly based on filenames and versions (such as packages). Another example is the requirement for an *SQL capable database* which can be satisfied by either MySQL or Posgres software.

- A development scenario is one where a library is needed for a F/OSS application. If the library is provided during the bundling phase by only one unique source, the dependency approach based on files would force the bundler to specify the name of the source providing the library. If a different source containing the library with its documentation is released later, the constructed bundle would not be able to detect that the new source library is usable. With a feature based approach, this problem is avoided, and the need for keeping track of changes of other projects is reduced as long as each distributed content unit clearly defines the features it provides and the features its needs.

An example of features that are pertinent to Project development phase are software Conventions. The scope of these Features is the project itself, and so they do not appear outside of the Project. Conventions can have many types, e.g., coding conventions, documentation conventions, design and format conventions, etc.

One of the features of ContentUnits is the License that the content unit respects. A ContentUnit must also specify a set of Features that it depends upon in order to be executable. Features can thus be used to describe units in a manner that facilitates location and distribution rather than relying on version numbers of specific InstallableUnits.

A Bundle groups ContentUnits for a given application purpose. It provides all the features provided by its ContentUnits plus specific Features provided by the Bundle itself. Examples of bundle specific features are the License that have to be compatible with the Licenses of the ContentUnits that compose the bundle or the Status that defines in which state the Bundle is (Beta, Release Candidate, Release, Stable...) A Bundle also defines the features that are needed by the ContentUnits it contains. This information is then used at the distribution level in order to compute dependencies.

Each InstallableUnit provides the features of the Bundle plus specific features like installation or compilation options. InstallableUnits know what features needed by the Bundle they encapsulate. This information can be used to compute dependencies without having to define needed InstallableUnits in advance. A License has to be chosen that is compatible with the License of the Bundle. Dynamic Features of InstallableUnits like performance (based on bug tracking for a given architecture for instance) or popularity can be computed. These can help both end-users and projects. On the one hand, end-users can detect potential issues and choose another InstallableUnit. On the other hand, project members can detect that an InstallableUnit is only used for a given set of Features being a small part of the whole set of provided Features. Thus the distribution process could be refactored based on the extracted knowledge and other lighter bundles only containing expected Features be built and distributed.

Current Steps

A useful exercise would now be to provide a simple formalism for the model, the goal of which is to express core concepts and all forms of content dependency relations in an unambiguous manner. This would be comprised firstly of the closed aggregate set *Units* that groups together ContentUnits, Bundles and InstallableUnits. The second aggregate set is *Features* which is an open aggregation whose predefined members include the sets *License*s and *VersionNumber*s.

All content dependency relations in F/OSS can be modelled using functions over *Units* and *Features*. A function entitled "features of" enumerates the features associated with a unit of content. Its signature is $Units \rightarrow \text{SetOf}(Features)$.

The inverse function $\mathcal{F}^{-1}$ yields a set of units with specific features. This function is crucial to locating units that satisfy given search criteria (specified as features). For instance, if the feature *crypto* with signature *sig* is being looked for, then the units that satisfy the request are $\mathcal{F}^{-1}(\{crypto\}) \cap \mathcal{F}^{-1}(\{sig\})$.

As an example, an *upgrade* relation can be modelled as an instance of a function $(Unit \rightarrow VersionNumber) \rightarrow (Unit \rightarrow VersionNumber^*)$. An instance $(u_0 \mapsto 1.0) \mapsto (\{\ u_1 \mapsto 1.1\}, \{u_2 \mapsto 2.0 \mapsto (u_3 \mapsto 2.0.1)\})$ means that a basic version of the unit u_0 was subject to a project split with two

upgrades u_1 and u_2. The unit u_2 is subsequently upgraded with unit u_3.

Patching also guards the relation between content units: $Unit \rightarrow Unit \rightarrow Unit$, where the entry $(u_0 \mapsto u_1 \mapsto u_2)$ means that unit u_0 was patched with the contents of unit u_1, and the result is u_2; again, a unit can be a content unit, bundle or installable unit. The goal is that, from the basic sets, each Project can define the content dependency relations important to it.

This is just preliminary. It suggests that one can abstract away from project distribution details when defining unit dependencies. For instance, upgrade and patching are not the same thing: the former is a type of logical update, the latter a physical update.