Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Michal Papež

SOFAnet 2

Department of Distributed and Dependable Systems

Supervisor: RNDr. Tomáš Bureš, Ph.D.
Study program: Computer Science, Software Systems

2011

I would like to thank to my supervisor for all the advice he gave me and for aiming my work on the most valuable topics out of the potentially broad scope of this thesis. I would also like to thank to my family and girlfriend for their support, tolerance and all the many little things that have brightened my days of work spent in front of a computer screen.

Title: SOFAnet 2
Author: Michal Papež
Department: Department of Distributed and Dependable Systems
Supervisor: RNDr. Tomáš Bureš, Ph.D.
Supervisor's e-mail address: bures@d3s.mff.cuni.cz

Abstract: The aim of SOFAnet 2, as a network environment of the SOFA 2 component system, is to exchange components between SOFAnodes in a simple and rational way. Current concerns of the SOFA 2 users about software distribution are analyzed and discussed. New high level concepts of Applications and Components are defined together with their mapping to SOFA 2 first class concepts, means of distribution and removal. Furthermore a methodology to keep SOFA 2 repository clean is introduced. All new elements as concepts and operations are studied using a formal set model. The proposed concept of SOFAnet 2 is proved by a prototype implementation.
Keywords: distributed systems, component systems, software distribution

Název práce: SOFAnet 2
Autor: Michal Papež
Katedra (ústav): Katedra distribuovaných a spolehlivých systémů
Vedoucí diplomové práce: RNDr. Tomáš Bureš, Ph.D.
e-mail vedoucího: bures@d3s.mff.cuni.cz

Abstrakt: Cílem SOFAnet 2 jakožto síťového prostředí komponentového systému SOFA 2 je jednoduchá a rozumná výměna komponent mezi jednotlivými uzly. Současné požadavky uživatelů SOFA 2 na distribuci software jsou analyzovány a diskutovány. Protože koncepty SOFA 2 nejsou navrženy pro účely distribuce, byly zavedeny nové koncepty pro aplikace a komponenty. Tyto nové koncepty jsou mapovány na koncepty SOFA 2 a studovány na formálním množinovém modelu. Pomocí tohoto modelu jsou definovány operace pro distribuci, instalaci a odstranění aplikací i komponent. Jsou také navržena pravidla, podle kterých je možné vyčistit repozitář SOFA 2 od nepotřebných částí software. Koncept SOFAnetu 2 je ověřen na prototypové implementaci.
Klíčová slova: distribuované systémy, komponentové systémy, distribuce software

Contents

1. Introduction

Various computer programs are a considerable part of our everyday life and the extend of using computers not only for work is continuously increasing. Moreover many complex computer systems are directing important processes as money transfers and water distribution remain unseen for most of us. The size of such systems is also growing rapidly, because the tasks that we performed by many stand alone systems are now desired to be integrated and performed by one coherent system. The new systems of various state bureaus are a good example. Therefore, the software design, development and maintenance are more complex and demanding. The software engineering is a designated engineering discipline which is concerned with all those aspects of the software life cycle.

1.1 Distributed Hierarchical Component Systems

Many systems, starting approximately from an information system for a middle-sized company, are seldom deployed on one machine. Their parts are distributed to allow balancing of the load of the computers, their CPU, memory, disk, networking usage and more. Often, companies have designated machines tailored for a particular task, such as database machines with super-fast raid arrays, computing machines with multiple CPUs, web-hosting machines with extended security and defense, etc.

When designing a large and complicated system, the overall mental complexity is very simplified when the system can be split into a number of logically separated parts. The logical and functional separation of parts is also crucial for development and maintenance, because it allows localization of modifications and concurrent work. Once those parts are treated as black-boxes that communicate with each other only using well defined interfaces, we are close to the level of abstraction that is necessary to develop large systems. Based on the communication protocol, the tightness of coupling and some other properties, we can call those black boxes components or services.

A component system is a facility that allows software development based on components. Each component system has a component model which defines basic relations, properties and behavior of all concepts the system operates with. Component based software engineering is an acknowledged way of developing large software systems, although its popularity is currently dropping in favor of service orientation.

Many companies have developed their proprietary component systems. For example, the Microsoft's COM and DCOM [1] were quite popular for developing applications for Windows around the year 2000, but it has been deprecated in favor of the .NET Framework [2]. Enterprise Java Beans [3] is still a popular component system to build enterprise applications.

Some systems were implemented according to a well known standard. For example

VisiBroker [5] is a commercial product implementing the CORBA Component Model [4] standardized by the Object Management Group. Many non-commercial and open source implementations exist too. Component models of these systems mostly do not offer advance features such as hierarchical composition of components or multiple communication styles. On the other hand, they have support guaranteed by their vendor and offer large plateaux of tools that were proven working on many large projects.

From the academic component system, we can name for example SOFA 2 [8] and Fractal [9]. Although the academic component systems offer many advanced features as those mentioned above, they are not widely accepted by industry. A discussion on their common shortcomings can be found in [14].

1.2 Software Distribution

Software distribution is not only related to trading with applications or software in general. In distributed environment, even a small part of software may be distributed among the nodes of single vendor. A typical example of a scenario that lowers the costs of developing large system is distribution of a component for reuse.

For the SOFA component system [17] a distribution framework called SOFAnet was worked out on a middleware level to a stage of prototype implementation [20]. It is focused not only on component distribution, but also on e-commerce support, sharing and searching. As the SOFA component system evolved into SOFA 2, a similar network environment for the new component system became desired.

1.3 Goals

The main goal of the thesis is to propose a way of exchanging components between SOFA 2 repositories and develop prototype implementation as a proof of concept. The way of distribution should be as rational as possible and allow the users to distribute what they really want to distribute. We prefer developing a rather simple and usable solution compared to proposing a complex system that can not be implemented to usable state in scope of a master thesis.

1.3.1 Current Concerns for SOFA 2 Users

We want to bring the most value to the users, therefore their current concerns are put under focus. We will focus on the fundamental use cases such as the following ones, because they are not sufficiently coved by the current SOFA 2 implementation:

- Store application or a set of components in a single file. Then install it from that file in another repository.

- Basic peer-to-peer distribution of applications or components.

- Application uninstallation.
- Keeping trash in a repository under control.

We feel that support for the above mentioned use cases should be provided sooner then we can start to think about corporate networks. The corporate networks are not the primary concern for the users yet, so they are dropped from the scope of this thesis. That means that we focus on distribution and uninstallations in more depth, but we omit licensing, sharing and searching networks proposed in the previous SOFAnet implementation for corporate intranets [19].

1.4 Structure of the Thesis

The *chapter 2: Context and Previous Work* describes briefly the SOFA 2 component system and provides background information about previous work related to the SOFAnet. The *chapter 3: Analysis* complements it with detailed analysis. The next *chapter 4: Model of the Solution* mathematically models the proposed solution using sets and *chapter 5: Implementation* provides description of the prototype implementation. The *chapter 6: Evaluation and Related Work* gives evaluation of the presented solution with respect to the goals of the thesis and the primary use cases (that are defined in *chapter 3: Analysis*). The overview of the related work is also presented in this chapter. The last *chapter 7: Conclusion and Future Work* gives a conclusion, as well as future work ideas.

2. Context and Previous Work

2.1 SOFA

SOFA (SOFtware Appliances) [6] is a research project introduced in [17]. SOFA is meant to be a component-based platform-independent architecture for software development (e.g. component system). The authors of [17] established the terms SOFAnode and the SOFAnet. The SOFAnode allows component storage, creation and execution, while the SOFAnet is a network interconnection of SOFAnodes that facilitates electronic trading with components.

After its introduction in [17], the SOFA project targetted mainly the structure of the SOFAnode. The SOFAnet was nearly left intact for five years until [18] and [19]. Its evolution began at almost the same time the main part of the SOFAnode, hand written template repository, was to be reimplemented [7]. The original SOFA project was soon after that abandoned in favor of its new version, the SOFA 2.

2.2 SOFA 2

The core features of SOFA 2 were introduced in [14] and [15]. In comparison to SOFA, SOFA 2 component model is defined using a MOF-based meta-model [10], which allows generation of model handling code. Therefore the SOFA 2 model can be modified with ease and can be more complex. Many new features were introduced. A new approach to implementation and extension of the controller part of components [16] is among the most important ones. The new controllers are micro-component based and a brand new micro-architecture meta-model was added to support them. They allow changes in component control chain similarly to AspectJ [11] approach. Also, component connectors became first class concepts.

The main resource of the SOFA 2 project related papers and documentation is [8].

2.3 SOFAnet

2.3.1 First thoughts of SOFAnodes and SOFAnet

The idea of connecting SOFAnodes into SOFAnet has been here for a relatively long time. In 1998, the idea of the functional parts of the SOFAnodes and how they should be connected into SOFAnet was published in [17]. SOFAnet was intended to allow both manual and automatic distribution (triggered by an event) of components, using both push and pull methods[1]. No implementation was mentioned.

[1]Pushing is a means of transfer that is initiated by sender. It is also called sending. Similarly, pulling (or downloading) is initiated by receiver.

2.3.2 SOFAnet Variant by Petr Panuška

Based on the view of SOFAnet in [17], the concept has been worked out by Petr Panuška [18] in depth.

Bundle

Panuška introduced a bundle as a packaging abstraction of the SOFA project. The bundle is a set of components with their implementations. It is realized as a JAR file [13] containing files with implementations of components and a manifest file that describes the content of the bundle.

Panuška's bundle could contain only one implementation of component[2], without including the required sub-components. Therefore, the bundle could contain something so small that it was almost no use by itself (especially when the required sub-components were missing).

On the other hand, the bundle could contain any arbitrary collection of implementations of components. We could include a few full-fledged applications inside such bundle too.

Bundle offer

A bundle offer is the second abstraction introduced by Panuška. Its purpose is to inform another SOFAnode what components are available for download and where. It contains a set of identifiers of components (name and version pairs), and each of the identifiers of components is accompanied with a name of node where the implementation of component can be obtained.

The implementation of the bundle offer is almost identical to the implementation of the bundle. All the important information is contained in the manifest file inside a JAR file. Therefore the only difference to bundle is, that the files containing implementations of components are omitted.

Use cases for SOFAnet

Panuška also described several use cases and business processes supported by the SOFAnet and SOFAnodes, namely:

- Making a Deal
- Sending patches for free
- Update downloading

The main scenario, Making a Deal, covers bundle creation, sending and installation and therefore shapes a base of distribution functions.

[2]Component implementation is a basic runtime unit of SOFA. It comprises of Architecture, Frame, Interface Types and optionally a code for primitive component.

2.3.3 SOFAnet Variant by Ladislav Šobr

A more ambitious and deeper vision of SOFAnet was presented one year later in [19] by Ladislav Šobr. He added licensing models and incorporated license into the structure of Panuška's bundle. Furthermore, he worked out a concept of three networks cooperating with each other and with SOFA runtime too.

Three Networks of SOFAnet

Distribution Network The Distribution Network primarily transfers bundles between SOFAnodes. It utilizes both the push and pull transfer models. It is targeted on the Internet.

Share Network The main goal of a share network is to share bundles among SOFAnodes and facilitate license management of the bundle in a corporate environment (on a subset of intranet). For each bundle it has a star topology with one share manager.

Search Network The search network facilitates searching for share managers over intranet. It transmits search queries and replies.

Capabilities of the Prototype Implementation

The prototype implementation was intended as a platform for future development and as a proof of concept. It could not be perfected because of the amount of work required.

It offered some interesting functions. For example bundles could be set to be handled automatically by triggers (on new bundle creation, arrival, sending, etc.).

When the SOFA runtime was missing a component, it could utilize the search network to look for it over corporate intranet. When a share manager of a bundle containing the missing component was found, it was asked whether the bundle could be obtained. If there were some free licences or the bundle could be uninstalled from some other SOFAnode, the bundle was transferred and installed on the node that was missing the component. This way, the missing component was obtained without any user intervention. The runtime could continue with instantiating the component or application transparently.

On the other hand, there was poor support for software removal among others. Only single bundle at a time could be removed and the dependencies between bundles and components in general were not handled by uninstallation functions at all.

The solution as a whole is relatively complex. There is a lot of structures and servers to be maintained and kept consistent with the central template repository, including the structures holding installed bundles, shared bundles, bundle offers, contracts, triggers etc.

2.3.4 Space for Improvement

Although the previous work on SOFAnet is extensive, there is still some room for improvement.

Evolution of SOFA

All the work on the SOFAnet mentioned above was done for SOFA. However, the SOFA evolved in SOFA 2 and everything, including the core interfaces, was redesigned and reimplemented. Not only the previous implementation might not be optimal according to the new SOFA 2 visions, but it is even unusable for SOFA 2 without huge modifications[3].

Unit of Distribution

The basic unit of distribution for all the previous implementations is the Panuška's bundle. As we have described above, in the extreme case, only an implementation of a single component is distributed in a single bundle. Such a bundle does not have much sense by itself. We feel that such a unit of distribution is too finely grained, because it does not reflect well what the users would normally like to distribute.

Fundamental Use Cases Not Covered

Also, we felt that the most frequent and fundamental cases of use are not sufficiently covered. For example:

Storing an application on a disk or even dumping a whole repository on disk.

Direct transfer of an application. It is currently troublesome to set up, and the unit of distribution is too finely grained.

Uninstallation that takes dependencies between bundles and components into account.

Keeping the repository clean is also desired. The users are currently concerned that there is a lot of garbage left in the repository when it is used for a longer period.

We would like to recreate the SOFAnet for SOFA 2 and focus on the fundamental scenarios as those mentioned above at the same time.

[3]It might be even faster to reimplement everything from scratch.

3. Analysis

3.1 Use Cases in Focus

3.1.1 Stake Holders

The *figure 3.1: Stake Holders* shows the basic Stake Holders intended to use SOFAnet 2 specific features. Node Administrator is a person who installs and uninstalls components and applications, configures the SOFA 2 node and takes care of deployed applications. An Assembler composes new components from subcomponents and also composes applications. The offerings of the new components and applications and also their acquisition are tasks performed by Distribution Managers. The Distribution Managers also set the distribution policies. In reality some of those roles may be represented by a single person. In an extreme case one person can even perform all of them.

Our set of the basic stake holders is smaller in comparison to [18]. We have omitted Users who only run applications and Developers who develop new components, because we are focused only on the features of the proposed SOFAnet 2 and not on the features of SOFA 2 in general.

3.1.2 Distribution

In this section we will describe the scenarios of usage that are related to distribution of software objects. In our case we will focus on applications and components. The basic distribution unit, that the system will use to distribute applications or their parts, will be called *package*[1].

Off-line Application Transfer

The *Off-line Application Transfer use case* describes how an application is transferred between two SOFA 2 nodes. The nodes do not need to be connected to the Internet. Any of the common means of large file transfer can be used to transfer a *package* containing the application.

Main successful scenario specification:

1a. Distribution Manager D1 offers an application to Distribution Manager D2. D2 accepts the offer.

[1]We decided to use the word “package” instead of the word “bundle”, because the latter would recall the *bundles* introduced in [18]. For the purpose of the analysis we do not want to fix ourselves to the idea of the Panuška’s *bundle*. Our use cases are different and there are some substantial changes in the model when comparing SOFA to SOFA 2. Later in the thesis we describe our reasons for using a different basic unit of distribution then Panuška’s *bundle*.

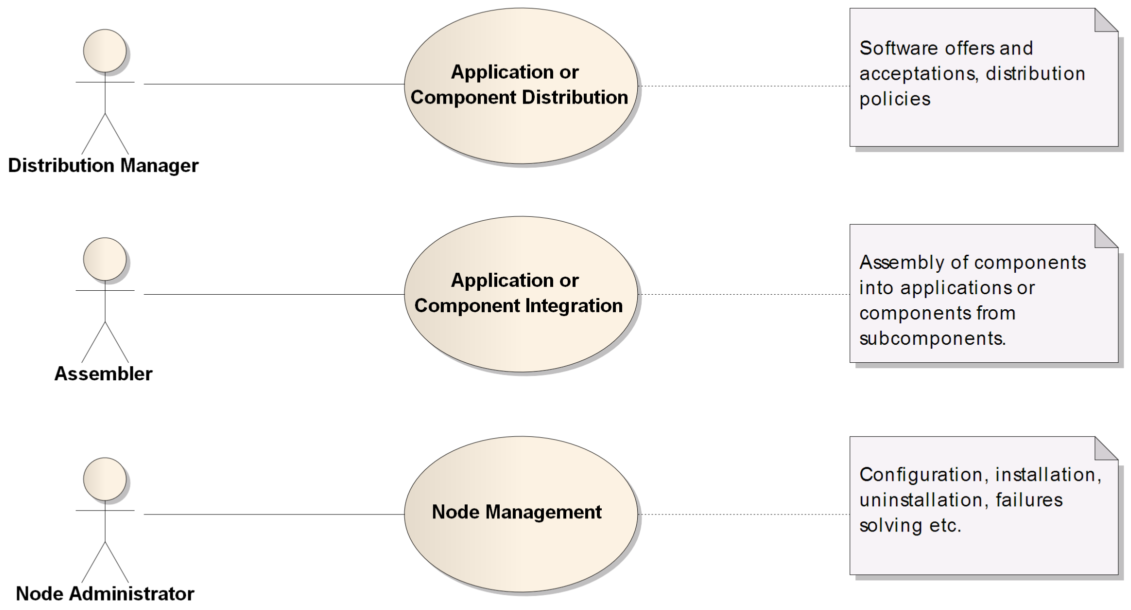

Distribution Manager
Application or Component Distribution
Software offers and acceptations, distribution policies
Assembler
Application or Component Integration
Assembly of components into applications or components from subcomponents.
Node Administrator
Node Management
Configuration, installation, uninstallation, failures solving etc.

Figure 3.1: Stake Holders

2a. Assembler A1 assembles a package file containing the application.
3. D1 sends the package using any common means of (possibly large) file transfer.
4. D2 receives the package.
5a. Node Administrator N2 installs the application packed in the package.

It is also desirable to have the possibility to transfer components to another SOFA 2 node for later reuse. The following variant of the *Off-line Application Transfer use case* describes the scenario.

Component transfer variant specification:

1b. D1 offers a set of components to D2 for reuse. D2 accepts the offer.
2b. Assembler A1 assembles a package file containing the set of components (including all subcomponents).

...

5b. Node Administrator N2 installs the components packed in the package.

On-line Application Transfer

The *On-line Application Transfer use case* describes how an application is transferred between two SOFA 2 nodes that are connected via Internet.

Main successful scenario specification:

1a. Distribution Manager D1 offers an application to Distribution Manager D2. D2 accepts the offer.
2a. Assembler A1 assembles a package containing the application.
3a. Node Administrator N1 installs the package on the SOFA 2 node of D2 using a direct connection.

The next variant is quite similar, the only difference is that it is concerned with components.

Component push variant specification:

1b. Distribution Manager D1 offers a set of components to Distribution Manager D2 for reuse. D2 accepts the offer.
2b. Assembler A1 assembles a package containing the set of components (including all subcomponents) and installs it on the SOFA 2 node of D2 using a direct connection.
3a. Node Administrator N1 installs the package on the SOFA 2 node of D2 using a direct connection.

The following two variants have the step where the Stake Holders make a deal in common with the main scenario and its first variant. However the actor that performs the transfer using the system under discussion (SuD) is different. The authors of [17] propose that the SOFAnet utilizes not only a push model, where the distributors of a bundle push it into the SOFA node of a customer. They propose also a pull model, where the customer "downloads" the bundle from the distributors SOFA node. Therefore the actors that initiate the transfer of the software parts in the following two variants are not the distributors.

Application pull variant specification:

1a. Distribution Manager D1 offers an application to Distribution Manager D2. D2 accepts the offer.
2c. Assembler A2 assembles a package containing the application from the SOFA 2 node of D1 using a direct connection.
3c. Node Administrator N2 installs the package (on the SOFA 2 node of D2).

Component pull variant specification:

1b. Distribution Manager D1 offers a set of components to Distribution Manager D2 for reuse. D2 accepts the offer.
2d. Assembler A2 assembles a package containing the set of components (including all subcomponents) from the SOFA 2 node of D1 using a direct connection.
3c. Node Administrator N2 installs the package (on the SOFA 2 node of D2).

3.1.3 Keeping Repository Clean

In *section 2.3: SOFAnet* we named a few articles that discuss the distribution of software between SOFA nodes. However, the software we obtained is getting outdated and eventually is replaced by a newer version. Our license contract may also expire. In such cases we also want to get rid the software that should be no longer used. Therefore software removal is needed for prospective future implementation of software licensing and sharing into SOFAnet 2. Another valid reason for software removal, probably the most frequent one, is that we just do not need it anymore.

Uninstallation

The *Uninstallation use case* describes how an application should be removed from a SOFA node. This task is important for Node Administrators, that are the only actors interacting with the SuD here.

Main successful scenario specification:

1. Node Administrator N1 asks the SuD to uninstall an application from a SOFA 2 node.
2. Node Administrator N1 is presented a list of software objects that will be removed from the SOFA 2 node.

3a. Node Administrator N1 confirms that the objects can be removed. The SuD removes them. All parts of other applications and software objects under development must remain in the SOFA 2 node and be functional. No useless parts of the removed application should remain.

Refine Objects to Remove variant specification:

3b. Node Administrator N1 wants to keep some of the objects. They tell the system which objects to keep.

4b. Node Administrator N1 is again presented with a list of software objects that will be removed from the SOFA 2 node.

5b. Node Administrator N1 confirms that the objects can be removed. The SuD removes them. All parts of other applications and software objects under development must remain in the SOFA 2 node and be functional. No useless parts of the removed application should remain.
(Same as *3a*)

Garbage Collection

SOFA 2 node can be accessed by various tools that allow creation or removal of first class entities. Those tools are intended to be the primary tools for Developers and Node Administrators, so these actors will always interact with the repository by means that go around the interface of SOFAnet 2.

We want to provide the Node Administrators with functions that clean the repository from unusable parts of software, because there always may be some software part left over during development or even normal operation no matter how we design and implement the SOFAnet 2.

The following *Garbage Collection use case* describes how a Node Administrator interacts with the system when they remove unusable software objects from a SOFA node.

Main successful scenario specification:

1. Node Administrator N1 asks the SuD to remove all useless software objects from a SOFA 2 node.
2. Node Administrator N1 is presented a list of software objects that will be removed from the SOFA 2 node.
   (Same as *point 2 of the Uninstallation use case*)

3a. Node Administrator N1 confirms that the objects can be removed. The SuD removes them. All parts of applications and software objects under development must remain in the SOFA 2 node and be functional. No useless software parts should remain.

Refine Objects to Remove variant specification:

3b. Node Administrator N1 wants to keep some of the objects. They tell the system which objects to keep.
(Same as *point 3b of the Uninstallation use case*)

4b. Node Administrator N1 is again presented with a list of software objects that will be removed from the SOFA 2 node.
(Same as *point 4b of the Uninstallation use case*)

5b. Node Administrator N1 confirms that the objects can be removed. The SuD removes them. All parts of applications and software objects under development must remain in the SOFA 2 node and be functional. No useless software parts should remain.
(Same as *3a*)

3.2 Addressing the Use Cases with SOFA 2 Concepts

SOFA 2 is a distributed system that exclusively uses a central repository to persist various concepts. We should focus on the concepts the repository stores first and then we can see if our use cases are sufficiently covered with the existing concepts.

3.2.1 First Class Concepts in SOFA 2

In the following paragraphs we briefly describe [2] nine first class concepts of SOFA 2.

Frame

Frame represents a view on a component from outside. It is a collection of provided and required interfaces. Every component has to implement a frame, even the top-level component implements frame with no interfaces. The components having the same frame are interchangeable. Therefore the frame acts to some extend as a component type.

Code Bundle

The Code Bundle represents an archive containing code for interface types, primitive components, connectors, etc. It can also be used to store application specific data.

Interface Type

Every interface, being a communication endpoint for a component, has to refer to an Interface Type object. The Interface Type object specifies interface implementation[3] stored in a Code Bundle.

Architecture

Each component is defined by an Architecture referencing the frame the component implements. There are two basic types of components: primitive and composite ones. Primitive components contain just the business code which implements the methods of their provided interfaces. Composite components consist of several sub-components and don't include any direct business code.

Assembly

The Assembly describes the structure of an application by a tree structure where each node holds the reference to the Architecture of a component. Architectures of sub-components are described by child nodes. This structure therefore describes how each of the composite components should be composed and also how the application should be composed from components.

[2]For more details please refer to [23]. Another comprehensible description of the main meta-model entities can be found at [21]. The micro architecture meta-model entities are nicely described in [24].

[3]The current implementation of SOFA 2 uses Java classes for that purpose. However, SOFA 2 is designed to be language independent.

Deployment Plan

Deployment Plans carry the information on where the components of the application will be deployed as well as the values of properties that will be passed to them at their startup.

Micro Component

Micro Components are elements of "micro-architecture meta-model". They are the basic building blocks of the SOFA 2 controller part. Each of them defines a collection of provided and required micro-interfaces. A Micro Component also has to specify its implementation. Therefore it plays a combined role of the Frame and Architecture together in the (normal) meta-model and introduces some additional functionality.

Micro Interface Type

The Micro Interface Type fulfils the same purpose as Interface Type in the SOFA 2 meta-model, but only for Micro Components.

Aspect

All Micro Components and control interfaces, that extend a controller part of a component, are put together in an aspect. The Aspect contains information about new control interfaces, instantiation of the Micro Components, and their connections.

3.2.2 Need for New High Level Concepts

From the list of first class concepts in SOFA 2 we can see that the concepts of applications and components are represented by Assemblies and Architectures. From the data model point of view, they provide us with all the information we need for both applications and components, including:

- name
- version
- previous version of the concept (if it is not brand new)
- complete structure of the sub-components where desired[4]

On the contrary the SOFA 2 concepts are not designed to support distribution in the first place. They are tailored for the purpose of software development and reuse, therefore they have much finer granularity. The use cases operate with applications or components like they are whole and can be used without obtaining and installing

[4]In case of Architectures, a sub-component can be defined two ways: by an Architecture or by Frame. It's up to developers how flexible they want the component Architecture definition to be. In case of Assemblies, we always have the full information about all the components used.

additional software parts. Therefore from the conceptual point of view we need to introduce something new to cover the use cases. The Assemblies and Architectures do not fit as they are.

3.2.3 Mapping of the New Concepts to the SOFA 2 Concepts

Most of the SOFA 2 entities[5] rely on other entities from the same repository. The kinds of dependencies between entities are derived from a SOFA 2 meta-model and from the choice of what will be a first class concept and what will not[6]. The choice of the first class concepts is important because they are persisted in the repository on their own, not as an owned part of any other concept. On the contrary the concepts of lower classes are stored with a first class concept and can not be persisted on their own.

In the *figure 3.2: SOFA 2 Meta-model* we can see various dependencies (denoted as arrows) between the main SOFA 2 entities. For example *Architecture* implements *Frame*. Some dependencies may be more difficult to see, but we also have to take them into account. A simple example is that a *Frame* depends on potentially many *Interface Types*, because it provides and requires *Interfaces* and each of the *Interfaces* has its *Interface Type*. *Interfaces* are not first class concepts, so they are stored as a part of a *Frame* in the repository.

The dependencies between the entities that are part of the SOFA 2 micro-architecture meta-model can be seen in the *figure 3.2: SOFA 2 Meta-model.* These entities and dependencies are not important for the distribution set of use cases, but we will need to track them when removing entities and keeping the repository clean.

Knowing about the direct and transitive dependencies between entities, we can introduce our concepts of Application and Component and map them to the SOFA 2 concepts.

Application

The most comfortable for user would be, if we could distribute a whole application including the SOFA 2 runtime and repository. Unfortunately there is no support for that in the runtime nor in the design of SOFA 2. Therefore we will try to distribute at least everything that is kept in the repository and fills the purpose of an application.

The closest to application from the point of launching are Deployment Plans. But the authors of SOFA 2 did not intend Deployment Plans for distribution, because they are of local nature.

The second closest is Assembly. It holds the information we want to know about an application and it allows us to generate Deployment Plans. Therefore it is by

[5]By the word "entity" we mean the first class concepts that are stored in a repository

[6]We have discussed the actual first class concepts in SOFA 2 in *section 3.2.1: First Class Concepts in SOFA 2* in more detail.

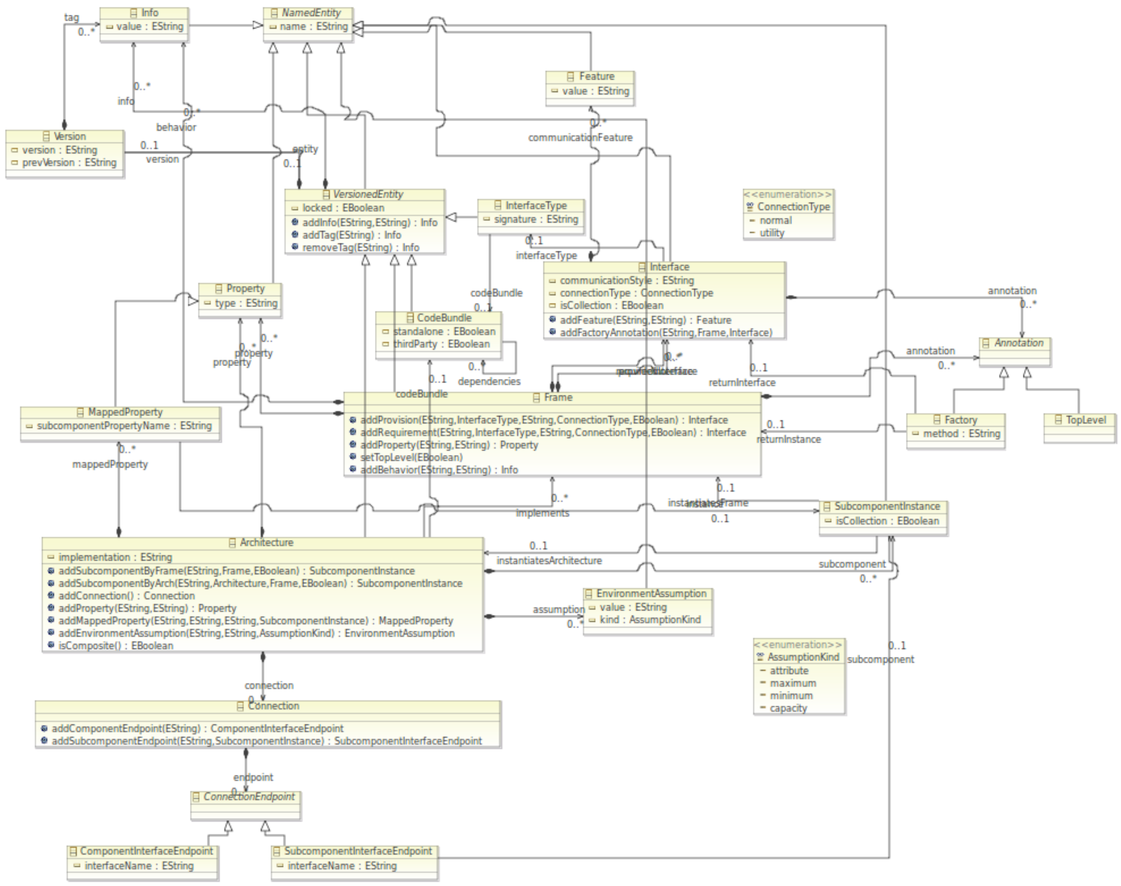

Info
value : EString
NamedEntity
name : EString
Feature
value : EString
Version
version : EString
prevVersion : EString
VersionedEntity
locked : EBoolean
addInfo(EString,EString) : Info
addTag(EString) : Info
removeTag(EString) : Info
InterfaceType
signature : EString
<<enumeration>>
ConnectionType
normal
utility
Interface
communicationStyle : EString
connectionType : ConnectionType
isCollection : EBoolean
addFeature(EString,EString) : Feature
addFactoryAnnotation(EString,Frame,Interface)
Property
type : EString
CodeBundle
standalone : EBoolean
thirdParty : EBoolean
Annotation
Frame
addProvision(EString,InterfaceType,EString,ConnectionType,EBoolean) : Interface
addRequirement(EString,InterfaceType,EString,ConnectionType,EBoolean) : Interface
addProperty(EString,EString) : Property
setTopLevel(EBoolean)
addBehavior(EString,EString) : Info
Factory
method : EString
TopLevel
MappedProperty
subcomponentPropertyName : EString
SubcomponentInstance
isCollection : EBoolean
Architecture
implementation : EString
addSubcomponentByFrame(EString,Frame,EBoolean) : SubcomponentInstance
addSubcomponentByArch(EString,Architecture,Frame,EBoolean) : SubcomponentInstance
addConnection() : Connection
addProperty(EString,EString) : Property
addMappedProperty(EString,EString,EString,SubcomponentInstance) : MappedProperty
addEnvironmentAssumption(EString,EString,AssumptionKind) : EnvironmentAssumption
isComposite() : EBoolean
EnvironmentAssumption
value : EString
kind : AssumptionKind
<<enumeration>>
AssumptionKind
attribute
maximum
minimum
capacity
Connection
addComponentEndpoint(EString) : ComponentInterfaceEndpoint
addSubcomponentEndpoint(EString,SubcomponentInstance) : SubcomponentInterfaceEndpoint
ConnectionEndpoint
ComponentInterfaceEndpoint
interfaceName : EString
SubcomponentInterfaceEndpoint
interfaceName : EString
tag
info
behavior
version
entity
communicationFeature
interfaceType
codeBundle
dependencies
annotation
returnInterface
returnInstance
property
mappedProperty
implements
instantiatesFrame
instantiatesArchitecture
subcomponent
assumption
connection
endpoint

Figure 3.2: SOFA 2 Meta-model
Originally published in [21].

SOFAnet 2 concept	SOFA 2 concept
Application	**Assembly** and all entities it requires: • Architectures • Frames • Interface Types • Code Bundles
Component	**Architecture** and all entities it requires: • Architectures • Frames • Interface Types • Code Bundles

Table 3.1: Mapping of the New Concepts to SOFA 2 Entities

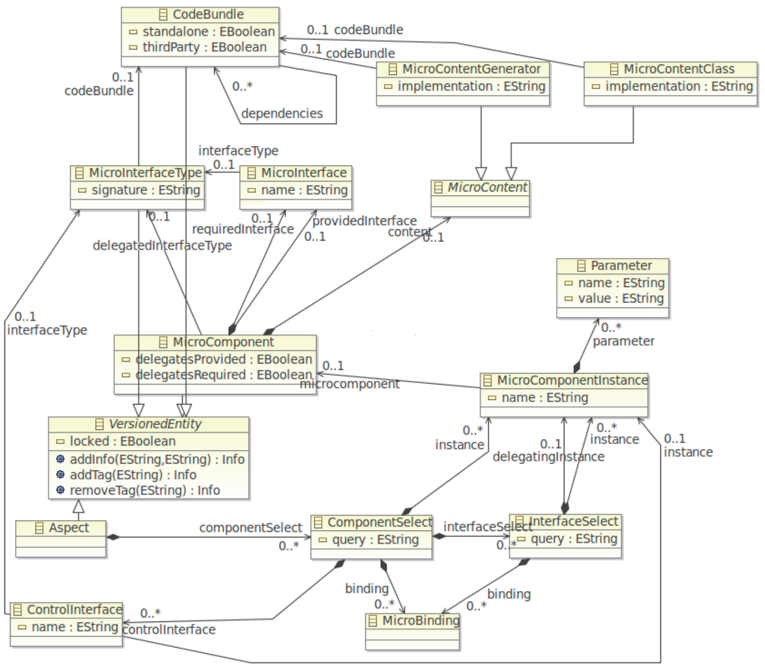


Figure 3.3: SOFA 2 Micro-architecture Meta-model
Originally published in [21], modified.

far the best candidate. However it is only a description of application structure, the implementation of the application is not a part of the Assembly as an entity. On the other hand, the Assembly holds the references on the entities the application should be composed of. It depends on them, on some of them directly, on some transitively. Therefore we can introduce an *Application*[7] as an Assembly and all entities it directly or transitively requires. The *table 3.1: Mapping of the New Concepts to SOFA 2 Entities* summarizes what kinds of entities will be inside the transitive dependency closure[8] of an Assembly.

It is common for applications to share some parts. Our *Applications* can share resources too, because they can share some entities. Anytime an *Application* would be distributed, it would be distributed with the shared entities, because they are part of the closure. When another *Application*, that is using particulary the same entities, is installed, the entities that are in common among the two *Applications* should be present exactly once in the repository.

Component

The closest SOFA 2 concept is Architecture. We introduce a *Component* as an Architecture and all entities that it directly or indirectly requires. The *table 3.1: Mapping*

[7]By the italic font face we will emphasize that we have a defined concept in mind.
[8]We will introduce the closure formally in the next chapter.

of the New Concepts to SOFA 2 Entities lists the entities that will be inside the dependency closure. The *Component* is introduced similarly to *Application*, therefore the previous paragraph about resource sharing holds true for *Components* too.

3.3 Methodology to Keep the Repository Clean

3.3.1 Which Entities Are Useless

In the *section 3.1: Use Cases in Focus* we have presented two use cases that address the removal of entities from the repository. The Actors of the use cases are concerned with which entities should remain in the repository. They want to keep everything that is important for them, but they want to get rid of the entities that they have no use for.

Using Applications

In the *section 3.2.3: Application* we have introduced a concept of *Application*, because we think it addresses our software distribution use cases, the best. The *Application* should be the most important concept for the users. However its concept does not cover Deployment Plans, because they are of local nature. But the users definitely want to keep them since they are used for launching of the *Applications*. From the *Applications* perspective this is all we need to keep:

- Everything we need to launch an application.
- Everything that is part of our *Application* concept.

Marking Entities under Development

The actors also want to keep what they are developing right now. All kinds of repository first class concepts can be under development at any given moment, so we can not succeed in introducing a concept with more coarse granularity. The smallest unit has to be an entity anyway. Therefore we propose marking of entities that are under development. The same dependency closures that we introduced for *Applications* and *Components* may be utilized. This way there would be no need to mark every entity that is part of current development. It should be enough to mark the entity that transitively uses all the others[9]. Maintaining a single entity mark or small set of marks in the worst case should be comfortable enough for the user, provided that the tools will implement reasonable default handling of the marks. This could for example involve automatic marking of every committed entity into repository to protect it from being accidentally considered a garbage.

[9]For example an Assembly Descriptor or a Deployment Plan.

3.3.2 What We May Want to Uninstall

In the *Uninstallation use case* the Node Administrator wants to remove an *Application*. The removal of an *Application* will probably take place most often. But the question is what other concept may the users want to remove? In the *section 3.3.1: Which Entities Are Useless* we were looking from a different perspective: What the users normally want to keep. But every piece of software we wanted to keep may be of no use anymore and we may want to remove it. Therefore there is another important object of uninstallation, a Deployment Plan. It is relatively easy for a Deployment Plan to become useless even though the Application it launches is still working. All it takes is to change the configuration of the SOFA 2 node by removing or renaming some docks.

Developers may also want to remove a whole part of an application they developed. In this case it is hard to determine automatically what should or should not be removed. Basically the unit is an entity here again. According to the *section 3.1: Use Cases in Focus* this is not our main concern. But we should provide an acceptable way. For example we can put in use the marks proposed in *section 3.3.1: Which Entities Are Useless.* That way we should obtain a solution that is more resistent to Node Administrators' errors in comparison to letting the Node Administrators remove the entities one by one by hand.

To summarize, the objects for which the uninstallation functions should be implemented are:

- Applications
- Deployment Plans
- Any entity in general. But this is not our main concern.

3.4 Non-functional Requirements

3.4.1 Ergonomics of the Solution

The users' content is one of our main concerns. When performing frequent tasks, our users will probably be interested in the simplicity and efficiency of the interface of tools. A transfer of an application is a good example of such a task. The interface should allow to transfer the application with the least effort, that means using only one click if possible.

For more advanced users or the users with higher demands on customizing the operations, a Command Line Interface (CLI) should be available. This interface should provide the user with the best possible control of the operations performed.

Only one GUI and one CLI should be needed, preferably integrated into the existing SOFA 2 tools as an extension. The new SOFAnet 2 tools have to be able to operate with the existing tools side by side. For example, no conflict should arise when a developer uses SOFA 2 IDE (without any extension) to develop an application and at the same time SOFAnet 2 CLI to obtain components for reuse.

3.4.2 Consistency and Concurrency in SOFA 2

The SOFA 2 repository can be accessed by multiple tools at any given instance. Certainly, our solution should not break the repository. Therefore all the actions the SOFAnet 2 tools will perform upon the repository should be somehow synchronized with the other tools or the repository should be accessed read-only. Furthermore any new information stored in the repository by SOFAnet 2 should be organized in such a way, that the other tools will still work as expected.

In case there will be any data stored outside the repository, the SOFAnet 2 tools should be able to work even when the repository is accessed by the tools that are not SOFAnet 2 aware and the external data are not updated accordingly.

Therefore we will design our solution in such a way that all information is stored solely in the repository if possible, because the repository is the only place designed for data persistence.

3.5 Requirements in Detail

In the early state of analysis a rather large catalogue of requirements was elaborated. We focused on a subset of the requirements gathered later. Because the catalogue is not an important point of this thesis, we present it in *appendix A: Catalogue of Requirements*. To put our first thoughts of all the desired features in contrast to the prototype implementation (see *chapter 5: Implementation*), we describe the status of the current implementation in the catalogue too.

4. Model of the Solution

In this chapter we will introduce a mathematical model of the proposed solution.

4.1 Repository as a Set

SOFA 2 repository stores various entities. First we need to introduce entity types, then we can introduce entities.

Definition 1. *Entity Type Set* is a set $\mathbb{T} = \{$ *Architecture, Aspect, Assembly, Code Bundle, Deployment Plan, Frame, Interface Type, Micro Component, Micro Interface Type* $\}$.

We will denote this set by $\mathbb{T}$. Its elements are called *Entity Types.*

The entities are uniquely identified by their name and version in the repository. The version itself contains information about all previous versions of the entity and is implemented so that it should be globally unique. Therefore the entities that have the same name and version are considered equal[1].

To model our solution, we need to calculate dependency closures for SOFA 2 entities, so we need the basic information for their computation. During runtime we are able to access the entities that an entity directly depends on. Also in the repository, each first class concept stores the information about its direct dependencies. We can make the direct dependencies part of our entity model, because this information is in fact contained in the first class entity and it has the same accessibility as the entity itself.

Definition 2. *Entity* is a quintuple $e = (n, v, t, D, I)$, where n is the entity's name, v is the entity's version, $t \in \mathbb{T}$ is the entity's type, D is a set of *entities* on which the entity directly depends and I is an (arbitrary) set of additional information.

We will denote a *set of all entities* by $\mathbb{E}$. We can then state that $D \subset \mathbb{E}$.

The purpose of a repository is to persist entities. Therefore it is sufficient for our model to understand a repository as a set of *entities.* We will not take such implementation details as locks on the repository and on particular entities into account. However, the set I in the definition of *entity* can be used as a base for extensions of our entity model. We introduced it to our definition not only because we will need it to recognize *entities* under development, but also because there is in fact something quite similar in the SOFA 2 meta-model.

In the *figure 3.2: SOFA 2 Meta-model* we can see that every *VersionedEntity* has any number of *Infos* and each of the *Infos* has a name and value (both are Strings). As long as any additional information we want to keep with an entity can be converted into a string or a small set of strings, this a good facility for storing it.

[1]For details on versioning and how versioned entities are manipulated in repository see Section 5.3. in [21]

Definition 3. *State of Repository* (in a given moment) is a set of entities, that are present in the repository (at that moment).

It will be usually denoted by R, obviously $R \subset \mathbb{E}$.

For convenience we will define the following set of auxiliary functions.

Definition 4. Let $e = (n, v, t, D, I)$ be an *entity*, $R \subset \mathbb{E}$ a set of entities. Then functions *dep*, *req*, and *type* are defined as follows:

- $req(e) = D$
- $type(e) = t$
- $dep(e, R) = \{d \in R : e \in req(d)\}$

The purpose of functions *req* and *type* is just to simplify the notation. The function *dep* is used to find out all entities from a given set, that are direct **dep**endants of a particular entity. It is meant to work in the opposite direction to function *req*, which returns a set of entity's direct **req**uirements.

4.2 Defining High Level Concepts

To define the new concepts of Applications and Components that we introduced in *section 3.2.3: Mapping of the New Concepts to the SOFA 2 Concepts* we need to define a closure of dependencies.

4.2.1 Dependency Closures

Requirements Closure

Definition 5. Let e be an *entity* and A a set of *entities*. Then function *requirements* is defined as follows:

- $requirements(e) = \{e\} \cup \left(\bigcup_{r \in req(e)} requirements(r)\right)$
- $requirements(A) = \bigcup_{a \in A} requirements(a)$

The *requirements* function computes the closure of dependencies iteratively from direct dependencies. Equivalently we could also define the *requirements* function upon an *entity* e using the following formula:

$$requirements(e) = \{e\} \cup requirements(req(e))$$

From this variant the implementation is straightforward using recursion.

Consistency of the Repository

The *Requirements Closure* can be also used to describe whether or not a repository is in a consistent state.

Definition 6. Let R be a *State of Repository*. We say that the *Repository is in Consistent State* iff[2] the following formula holds true:

$$requirements(R) \subseteq R$$

We can also say that R is *consistent*.

The definition says in other words that all requirements for all entities from the repository have to be in that repository too.

Dependants Closure

Definition 7. Let e be an *entity*, R and A sets of *entities*. Then function *dependants* is defined as follows:

- $dependants(e, R) = \{e\} \cup \left(\bigcup_{d \in dep(e,R)} dependants(d, R) \right)$
- $dependants(A, R) = \bigcup_{a \in A} dependants(a, R)$

The *dependants* function works in the opposite direction to the *requirements* function. It will come in use when we reason about entities that should be removed with an *Application*, *Component* or with an *entity* in general. Again, the *dependants* function upon an *entity* e and a set of *entities* R can be equivalently defined using the formula:

$$dependants(e, R) = \{e\} \cup dependants(dep(e, R))$$

However, the implementation of this function will be a bit more complicated if we take speed into consideration. It might be useful to cache the results of the $dep(e, R)$ somehow.

4.2.2 Application

We define an Application according to the notion presented in *section 3.2.3: Application.*

Definition 8. Let a be an *entity* and R a set of *entities*. An *Application* is a pair (a, R), where $type(a) = Assembly$ and $requirements(a) = R$.

We will call a the *main entity* of the *Application*.

[2] if and only if

4.2.3 Component

Similarly, we define a *Component* according to the *section 3.2.3: Component*.

Definition 9. Let c be an *entity* and R a set of *entities*. A *Component* is a pair (c, R), where $type(c) = Architecture$ and $requirements(c) = R$.

We will call c the *main entity* of the *Component*.

4.2.4 Package

Our use cases (see *section 3.1: Use Cases in Focus*) require the SOFAnet 2 *package* to hold either a set of *Components* or a single *Application*. We do not want to be too restrictive, because we think an abstraction is more than appropriate here. From our point of view, the package should contain an arbitrary set of important concepts and everything those main concepts need for their proper functioning in order to be reasonable for users. Therefore we will define a *package* in this more abstract sense.

Definition 10. Let M and R be sets of *entities*. A *Package* is a pair (M, R), where $requirements(M) = R$.

The set M is called the set of the *main entities*.

We can easily see that the ability of a *package* to hold *Components* and *Applications* is not compromised by our definition. For each *main entity* m that is an $Assembly$, there is a whole Application in the package (obviously $m \in M \implies requirements(m) \subseteq requirements(M)$). The same is true for *Components*.

For convenience we will define some auxiliary functions on packages too.

Definition 11. Let $p = (M, R)$ be a package. Then the functions *contents* and *mains* are defined as follows:

- $contents(m) = R$
- $mains(m) = M$

4.3 Semantics of the Use Cases

In this section we will describe the semantics of the use cases introduce in *section 3.1: Use Cases in Focus*. We will focus on the derivation of the new state of the repository in case of a successfully finished use case scenario. For clarity and simplicity we will not reason about the repository states in cases of scenario interruption[3].

[3]In all our use cases the modification of the repository contents happens in the last step. Therefore we do not miss any possible repository state caused by scenario termination before the last step. We leave the solution of a possible premature termination during the last step (like hitting a cancel button or killing the process in extreme case) on implementation.

4.3.1 Operations with Repository

To describe what happens to the content of a repository we need to introduce a notation that describes the change. Assuming we have two states of the same repository, R_{before} and R_{after}. R_{before} describes the state of the repository before a particular action and R_{after} after it. It is common in literature to note the immediate transition as $R_{before} \rightarrow R_{after}$.

This is useful when we want to emphasize the order of states and their relations. However, for our description semantics of the use cases, we will usually use only two states for each repository. We do not want to use more because we want to abstract from *the intermediate states* during the process updating the repository contents. On the other hand we want to describe the exact relation between the former and the new *State of the Repository* and some other inputs if they apply.

For this purpose we find a notation that uses functions more appropriate.

Definition 12. An *Operation with Repository* is a function f with at least one parameter p_0 where the first parameter p_0 is a *State of Repository* and the codomain of f is the set of all *States of Repositories*.

If R_{before} and R_{after} are two *States of the Repository* R, $p_1, \ldots, p_n$ are function parameters and $n \in \mathbb{N} \cup \{0\}$, we can denote:

$$R_{after} = f(R_{before}, p_1, \ldots, p_n)$$

We say that R_{before} is the *initial state* before the operation f and R_{after} is the *final state* after it.

4.3.2 Off-line Application Transfer

The steps *1a*, *3* and *4* of the *Off-line Application Transfer use case* do not require any user interaction with the system, therefore they are not important from the semantics point of view. We need to model the *package* creation described in step *2a* and *package* installation described in step *5a*.

Definition 13. *Package Creation Function* is a function *create_package* defined for any set of *entities* M and a *State of Repository* R as follows:

$$create\_package(M) = \begin{cases} (M, requirements(M)) & \text{if } M \subseteq R \text{ and } R \text{ is } consistent \\ \text{undefined} & \text{otherwise} \end{cases}$$

We can see that the codomain of the *create_package* function is a set of all *packages*, because its value is defined according to the *definition 10.*

Definition 14. *Package Installation* is an *Operation with Repository* that is defined as a function *install_package* for any *State of Repository* R and a *package* p as follows:

- R is *consistent*: $\quad install\_package(R, p) = contents(p) \cup R$
- undefined otherwise

Theorem 1. In cases where the *Package Installation* operation is defined, it results in a the repository being in *consistent state.*

Proof. Let R_{before} and R_{after} be two *States of the Repository* R and p a *package*, where $R_{after} = install\_package(R_{before}, p)$. For any *entity* $e \in R_{after}$ the following holds true:

- If $e \in R_{before}$ then $requirements(e) \subseteq R_{before}$, because R_{before} was *consistent.* From the *definition 14* we can see that $R_{before} \subseteq R_{after}$. Therefore we can conclude that
$$requirements(e) \subseteq R_{after}$$
- If $e \notin R_{before}$ then $e \in contents(p)$, which means
$$requirements(e) \subseteq requirements(mains(p))$$
$$requirements(e) \subseteq R_{after}$$

We can see that $\forall e \in R_{after} : requirements(e) \subseteq R_{after}$. According to the *definition 6* we conclude that
$$requirements(R_{after}) \subseteq R_{after}$$
which means that the repository is in *consistent state.* □

By now we have defined everything we need to define the semantics of the *Off-line Application Transfer use case*.

Semantics of the Off-line Application Transfer Use Case

Inputs:	S	-	Consistent state of the source repository.
	T_{before}	-	Consistent state of the target repository.
	M	-	A set of *main entities*, $M \subseteq S$, chosen by Assembler A1.
Results:	p	-	The package the Businessman B1 sends to Businessman B2.
	T_{after}	-	Consistent state of the target repository after a successful scenario.
Semantics:	p	=	$create\_package(M)$
	T_{after}	=	$install\_package(T_{before}, p)$

4.3.3 On-line Application Transfer

The *On-line Application Transfer use case* does nothing special in comparison to the *Off-line Application Transfer use case* from the repository point of view. In both cases some software parts are taken from one repository and installed into another repository. Of course the actors initiating the changes in the system are in some points different. But the characteristics of actor should have no impact on the semantics of the action, assuming the action is given exactly the same inputs. Therefore we can define semantics of the *On-line Application Transfer use case* right away.

Semantics of the On-line Application Transfer Use Case

Inputs:	S	-	Consistent state of the source repository.
	T_{before}	-	Consistent state of the target repository.
	M	-	A set of *main entities*, $M \subseteq S$, chosen by either Assembler A1 or Assembler A2.
Results:	T_{after}	-	Consistent state of the target repository after a successful scenario.
Semantics:	T_{after}	$=$	$install\_package(T_{before}, create\_package(M))$

We can see there is only one point of difference between the semantics of the two use cases: There is no *package* among the (intermediate) results. Otherwise the state of the repository T_{after} is obtained the same way. This is indeed what we want, because it is reasonable to obtain the same repository state no matter how we transfer the package.

4.3.4 Making Order in a Repository

In the step *2* of the *Uninstallation use case* the Node Administrator is presented a list of software objects, that will be removed from the SOFA 2 node. Therefore the list of objects (SOFA 2 entities seem to be fitting for this purpose) should be among intermediate results of the uninstallation. The kinds of objects the users usually want to keep in the repository and those they may want to remove were discussed in *section 3.3.1: Which Entities Are Useless*. We will now introduce a few functions that will concretize the ideas presented there.

Entities We Usually Want to Keep

Definition 15. *Entity under Development* is an *Entity* $e = (n, v, t, D, I)$, where $\star \in I$. The element $\star$ is called the *Development Mark.*

Definition 16. Let $e = (n, v, t, D, I)$ be an entity and $R \subset \mathbb{E}$ a set of *entities.* Then functions $info$, *assemblies*, *deployment_plans*, and *developed* are defined as follows:

- $info(e) = I$

- $assemblies(R) = \{\forall a \in R : type(a) = Assembly\}$
- $deployment\_plans(R) = \{\forall d \in R : type(d) = Deployment\ Plan\}$
- $developed(R) = \{\forall d \in R : \star \in info(d)\}$

Definition 17. Let R be any *State of Repository*. Then the function $entities2keep$ is defined as follows:

$$entities2keep(R) = requirements(\ assemblies(R) \cup deployment\_plans(R) \cup developed(R)\ )$$

The the *definition 17* defines a function that returns all the entities that should be kept in the repository according to the ideas presented in *section 3.3.1: Which Entities Are Useless*. All applications are kept, because an application is defined as an *Assembly* and its *requirements* closure. All *Deployment Plans* and *Entities under Development* are kept too with all the entities they (directly or indirectly) require.

Entities That Are Becoming Useless

The function $entities2keep$ is tightly related to the function that calculates the entities that should be uninstalled with a given entity. Any time we want to remove an entity e we need to remove the $dependants(e)$ too to keep the repository consistent. In general, everything we remove needs to be removed with all entities that depend on it.

When we remove a top entity of some concept, for example an Assembly or Deployment Plan, we would like to remove everything that will become useless once the top entity is removed too. Therefore, we would like to remove $requirements(e)$ if possible, and then also $dependants(requirements(e))$ because of repository consistency. In other words we would like to remove everything that the top entity needed, since it is potentially useless, and also everything that needed those entities in order to keep the repository consistent.

However, applications share components. It might happen, that we can not remove all the $requirements(e)$, because any other application than e may be using some entity from it. The same is true for the Deployment Plans and it is even true for entities in general. Therefore we need to subtract $entities2keep(R \setminus dependants(e))$ from the list of entities scheduled for removal in order to keep all other *Applications*, *Deployment Plans* and the *Entities under Development* working.

Trouble with Relating Applications to Deployment Plans

There is one problem that can be seen only when the SOFA 2 meta-model is carefully examined. The *Assembly Descriptors* are not directly connected in the meta-model with the *Deployment Plans* that were constructed from them. How can we figure out which *Deployment Plans* was created from which *Assembly Descriptor* then?

The only way (beside of introducing changes into the meta-model) seems to be to compare the structure described by a particular *Assembly Descriptor*[4] to structures of all *Deployment Plans*[5] and see which *Deployment Plan* fits. If we would like to introduce such mechanism into our model of solution, we would need to work not only with SOFA 2 first class entities, but more likely with all its entities in order to stay on a consistent level of abstraction. That would blow our model many times in size, break its simplicity and the chances of reuse.

We decided to keep using only the first class entities in our model of solution. Therefore we sacrifice some precision in the cases where a *Deployment Plan* needs to be removed due to an *Application* removal. In such cases, the entities that are only part of a *Deployment Plan* but not part of the *Application*, may not be properly uninstalled. However, this should be rare and it affects only the highly deployment-specific entities as *Aspects* and *Micro Components*. Moreover, this compromise will only affect the Uninstallation function.

Definition of the Function Computing The Entities to be Removed

Definition 18. Let R be any *State of Repository* and e an entity. Let K be a set of entities, that satisfies:

$$K = \begin{cases} dependants(e) \cup deployment\_plans(R) & \text{if } assemblies(dependants(e)) \neq \emptyset \\ dependants(e) & \text{otherwise} \end{cases}$$

Then the function *entities2remove* is defined as follows:

$$entities2remove(R, e) = dependants(requirements(e)) \setminus entities2keep(R \setminus K)$$

The set K in the *definition 18* is a set of entities that lose their protection by the *entities2keep* function. In other words, these are the concepts that do not need to keep working after the removal of an Uninistallation of an *entity* (or *Application*, *Component*, ...). Certainly, if we remove the *entity* e, we do not insist on keeping it working. But we can not keep the *entities* that are *dependant* on it too, otherwise the repository will be inconsistent. Therefore the $dependants(e)$ are always inside the set K.

[4]The *Assembly Descriptor* describes the structure of an application by a tree structure of *Instance Assembly Description* nodes connected by *Top Level Instance* and *Sub Component Instance* relations. The nodes hold the references to *Architectures* of all *Components* and their Subcomponents too. Therefore this structure describes how each of the composite *Components* should be composed and also how the *Application* should be composed from *Components*.

[5]The Deployment Plan describes the structure of an application to launch by a tree structure of Instance Deployment Description nodes, that are connected by *Top Level Instance* and *Sub Component Instance* relations too. However the nodes have a different type than those in case of an *Assembly*. They hold more references: *Architectures* to be instanced and *Code Bundles* to be loaded.

In the first case of the definition of K, the $deployment\_plans(R)$ are also included in the set K. We need to include the *Deployment Plans* that deploy an *Application* we need to remove. Otherwise only the *Assembly* on the top of the *Application* would be removed, because all the components used by the *Application* are referenced from its *Deployment Plan*. We think this is not the behavior users would expect. We have to include all the *Deployment Plans* from the repository and not just some of them, because of the problem of relating the *Applications* to *Deployment Plans*.

By the inclusion of all *Deployment Plans* from the repository we can not break any *Deployment Plans* that are not related to the *entity* we remove. Only the entities that are in $dependants(requirements(e))$ may be removed. Only those *Deployment Plans*, that require an entity to be removed, are removed. But all the entities that are required for the *Applications* we keep are kept too. Therefore all the *Deployment Plans* deploying the *Applications* we keep will also remain.

If there was such a mechanism that could tell us which *Deployment Plans* were created for the *Applications* we want to remove, we would add only those *Deployment Plans* to the set of “not defended” entities K.

4.3.5 Uninstallation

We are now equipped with the precise definition of a function that computes the set of entities that should be removed during uninstallation. We can now proceed to the definition of the Uninstallation Operation and to the semantics of the *Uninstallation use case*.

Definition 19. *Uninstallation* is an *Operation with Repository* that is defined as a function *uninstall* for any *Consistent State of Repository* R and an *entity* e as follows:

$$uninstall(R, e) = R \setminus entities2remove(R, e)$$

Theorem 2. The *Unistallation Operation* results in a repository being in a *consistent state.*

Proof. In the final state of the repository after the operation, every entity needs to have all its requirements in the repository too. Otherwise the repository is not in a *consistent state*. We remove only the entities given by the $entities2remove$ function value. Therefore it is enough that each entity from the set returned by the $entities2remove$ function is in the returned set together with all its dependants.

The $entities2remove$ function can be rewritten as

$$entities2remove(\dots) = dependants(\dots) \setminus requirements(\dots)$$

because the $requirements(\dots)$ computed inside the $entities2keep$ function are subtracted from the $dependants(\dots)$ inside the $entities2remove$ function.

Let's examine the result of the *entities2remove* function. Let D be any set returned from the *entities2remove* function. For a proof by contradiction, let e_1 and e_2 be two entities, where $e_1 \in D$, $e_2 \in dependants(e_1)$ and let's assume $e_2 \notin D$. This is the only case the repository consistency condition would break.

$$e_1 \in D \implies e_1 \in dependants(\dots) \implies e_2 \in dependants(\dots)$$

$e_1 \in D$, so it was returned from the $dependants(\dots)$ closure. Obviously, e_2 was also returned from $dependants(\dots)$, because $e_2 \in dependants(e_1)$.

$$e_2 \in dependants(\dots) \wedge e_2 \notin D \implies e_2 \in requirements(\dots)$$

e_2 was also returned from $dependants(\dots)$ and it was not in the result, so $e_2 \in requirements(...)$, otherwise it were in D.

$$e_2 \in dependants(e_1) \implies e_1 \in requirements(e_2)$$

We know that $e_2 \in dependants(e_1)$. The closures work in opposite directions, so $e_1 \in requirements(e_2)$. Furthermore we can deduce that:

$$(e_2 \in requirements(\dots)) \wedge (e_1 \in requirements(e_2)) \implies e_1 \in requirements(\dots)$$

$$e_1 \in requirements(\dots) \implies e_1 \notin D$$

If $e_1 \in requirements(\dots)$, it would have been subtracted from the result of $dependants(\dots)$. Therefore $e_1 \notin D$ and our assumption breaks. $\square$

We can conclude that the condition for the repository consistency can never break as a result of *Unistallation Operation*.

Semantics of the Uninstallation Use Case

Inputs:	e	-	Top entity of the concept to be uninstalled.
	R_{before}	-	Consistent state of the repository.
Results:	U	-	A set of entities scheduled for removal. It is presented to the Node Administrator during steps *2* and *4b* of the *Uninstallation use case*.
	R_{after}	-	Consistent state of the repository after a successful scenario.
Semantics:	U	$=$	$entities2remove(R_{before}, e)$
	R_{after}	$=$	$R_{before} \setminus U \ = \ R_{before} \setminus entities2remove(R_{before}, e)$

We have proved the repository will be in a consistent state after the *Uninstallation Operation* and therefore after a successful scenario of the *Uninstallation use case*. But there is still one open question about the entities that are left in the repository: May the *Uninstallation Operation* leave in the repository any "useless" entities? We will understand the "useless" entities as the entities that are not in the result of the *entities2keep* function[6].

It may introduce some garbage in the repository in cases of *Application* removal, because of the problem with *Deployment Plans* and *Assemblies* relation, that were discussed in *section 4.3.4: Making Order in a Repository*. We think the most common cases are those where the *Deployment Plans* are not using the entities that are not part of the *Application* (like *Micro Components* and *Aspects*[7]). In cases where only such Deployment Plans are removed (as a result of removing *Application*), no new garbage will be introduced.

If there was such a mechanism that could tell us which *Deployment Plans* were created for an *Application* we need to remove, the cleanest solution would be to uninstall those *Deployment Plans* prior to the uninstallation of the *Application*.

We suggest that the tools that perform the *Uninstallation Operation* also offer the user a default option to trigger a Garbage Collection after the *Uninstallation* is finished. This will clean all the garbage that may be in the repository before the *Uninstallation* or that might be introduced by it. The semantics of the Garbage Collection follows in the next section.

4.3.6 Garbage Collection

Semantics of the Garbage Collection Use Case

Inputs:	R_{before}	-	Consistent state of the repository.
Results:	G	-	A set of entities scheduled for removal. It is presented to the Node Administrator during the steps *2* and *4b* of the *Garbage Collection use case*.
	R_{after}	-	Consistent state of the repository after a successful scenario.
Semantics:	G	$=$	$R_{before} \setminus entities2keep(R_{before})$
	R_{after}	$=$	$entities2keep(R_{before}) \; = \; R_{before} \setminus G$

Process of the Garbage Collection

According to the semantics we have just presented and the *Garbage Collection use case*, the action could be implemented in the system as follows:

1. The set of garbage will be found using the formula presented in the semantics:

$$G = R_{before} \setminus entities2keep(R_{before})$$

[6]We will define the semantics of Garbage Collection in the *section 4.3.6: Garbage Collection* in this sense.

[7]The exception are the internal entities. But internal entities should be excluded from all repository operations and never be considered "useless".

2. The set of garbage will be presented to the user.
3. They decide, if it should be removed as a whole, if some entities should be marked as *Entities under Development* or if they just want to ignore the garbage and cancel the operation.
4. If the user agrees to remove the garbage as a whole, it is removed. Otherwise if the marks on entities have been updated, the process should be invoked again from point 1 (rather on demand).

4.4 Distribution Package Implementation

The implementation of a Distribution Package[8] is important on a different level of abstraction than the implementation of tools and actions. The Distribution Package implementation is playing a role of an interface, because it should not be dependent on the implementation of the tools, but all the tools should use it. It should be also resistent to various changes in SOFA 2 in general, even in its meta-model if possible.

4.4.1 Entity Package

In the *definition 10* we have proposed an abstraction of the distribution package. Our model of the solution operates on entities contained in the abstracted distribution package. Therefore we think the entities contained in the package should be accessible randomly, one by one by the tools that implement the functions related to the distribution packages.

The same rules about resistance to changes should apply to how the entities should be stored inside the distribution package. We decided to see this representation of an entity as another package called *Entity Package*, because even though the distribution of single entities is not our main concern, it should still be possible.

Entity Package Implementation

In order to maximize the independence of the entity package on the implementation of the SOFA 2 tools, it was necessary to use the basic entity description the tools are using for entity creation in the package. The basic information about the entity is by default stored in a few files (that the tools can operate), therefore the entity package is implemented[9] as a zip archive. Because it contains a single entity, the name of this

[8]From this section onwards, we will need to distinguish two types of packages. One is the *package* discussed earlier. We will call it a Distribution Package in the rest of this chapter to emphasize its application or component distribution purpose. The other type of a package is an Entity Package, which will be discussed in the next subsection.

[9]The implementation of the entity package was first described in [22] by the author of this thesis. However, in this early stage of SOFAnet 2 development the two types of packages were not distinguished.

archive was fixed to *<name of the entity>-<version of the entity>.sp*. This naming convention tells the tools which entity is inside the package without the necessity to extract the archive.

The archive contains two or three files:

adl.xml Definition of the entity described by SOFA 2 architecture description language file. See Part IV, SOFA 2 ADL Developer Guide in [22] for SOFA 2 ADL description.

contents.xml Contains name, version, type and tags of the entity and lists all its dependencies by their name and version (see *listing 4.1: Entity Package contents.xml*). The `contents.xml` file is meant to be processed by tools first, because they should check that all the dependencies of the entity are satisfied.

<name of the entity>.jar Contains the code of the entity. Present only if the entity has some.

```
<?xml version="1.0" encoding="UTF-8"?>
<sofa_package>
  <name>org.objectweb.dsrg.sofa.examples.logdemo.assm.LogDemo</name>
  <version>4a87f0d8515fa5cdd81ae0e1626e4e3769220594</version>
  <type>assembly_desc</type>
  <tags>
    <tag>current</tag>
  </tags>
  <dependencies>
    <depends>
      <name>org.objectweb.dsrg.sofa.examples.logdemo.arch.LogDemo</
         name>
      <version>574e0ec44fb2c1a09e246d829bd6cfd8c5b070a5</version>
    </depends>
    ...
    <depends>
      <name>org.objectweb.dsrg.sofa.examples.logdemo.arch.Logger</
         name>
      <version>da1a752b079a3d30a18b042ac3c20f4d018aa832</version>
    </depends>
  </dependencies>
</sofa_package>
```

Listing 4.1: Example of entity package `contents.xml` file

4.4.2 Composition of Distribution Package

The distribution package is composed of several files stored in a zip archive. Its structure is partially similar to the structure of an entity package.

The archive contains two types of files:

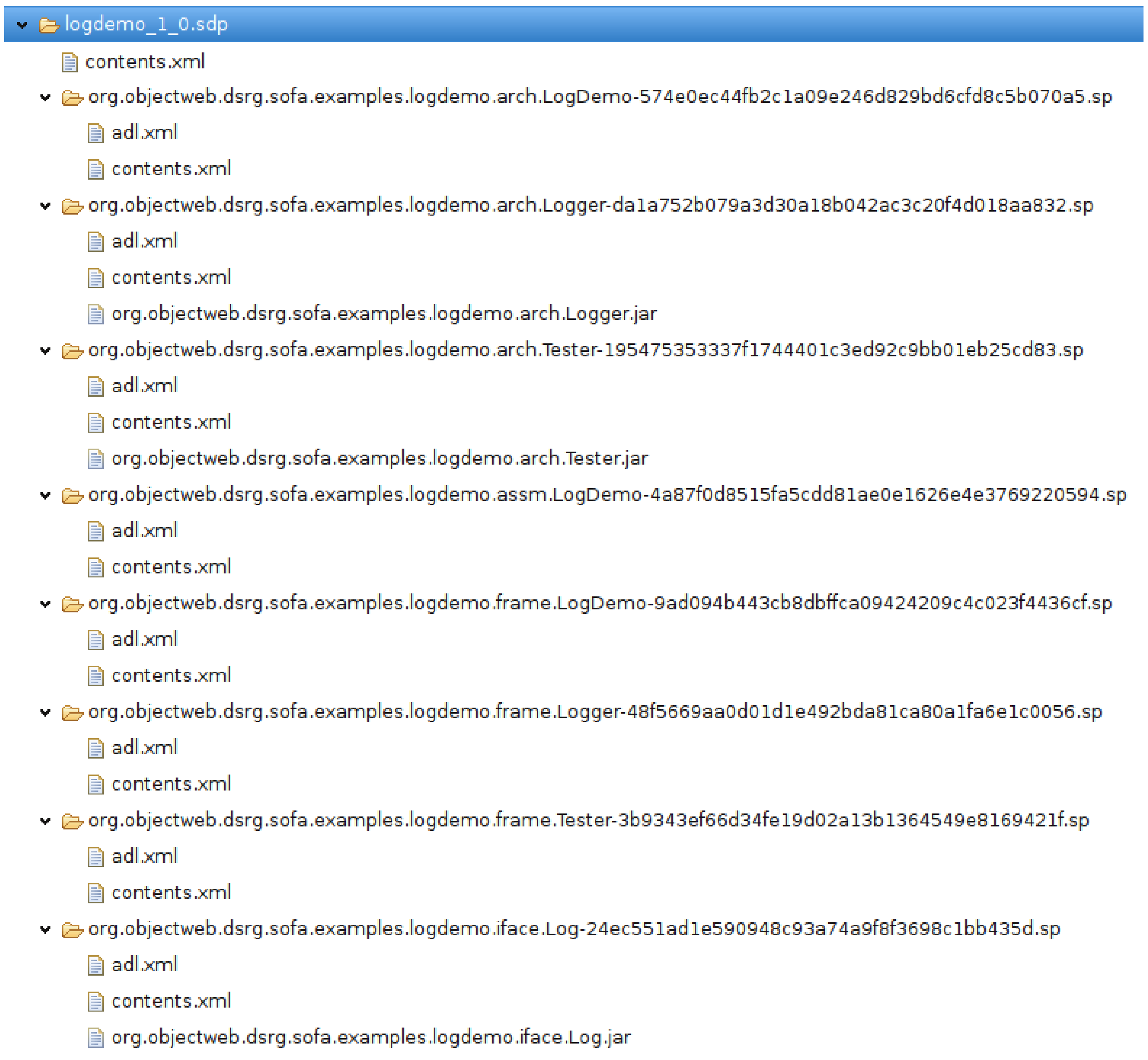
contents.xml Contains the name, version and type of the distribution package. For all main entities it lists their name, version and type. Furthermore all entities contained in the package are specified by their name and version. See *listing 4.2: Distribution Package contents.xml* for example. The contents.xml file is meant to be processed by package handling tools first. It should avoid the necessity to extract the contained entities in order to just examine the package.

<name of the entity>-<version of the entity>.sp Entity Packages. There is one such file for each entity contained in the distribution package.

```
<?xml version="1.0" encoding="UTF-8"?>
<sofanet_distribution_package>
  <name>logdemo</name>
  <version>1.0</version>
  <type>Application</type>
  <main_entities>
    <main_entity>
      <name>org.objectweb.dsrg.sofa.examples.logdemo.assm.LogDemo</
         name>
      <version>4a87f0d8515fa5cdd81ae0e1626e4e3769220594</version>
      <type>assembly_desc</type>
    </main_entity>
  </main_entities>
  <contained_entities>
    <contained_entity>
      <name>org.objectweb.dsrg.sofa.examples.logdemo.frame.Tester</
         name>
      <version>3b9343ef66d34fe19d02a13b1364549e8169421f</version>
    </contained_entity>
    ...
    <contained_entity>
      <name>org.objectweb.dsrg.sofa.examples.logdemo.assm.LogDemo</
         name>
      <version>4a87f0d8515fa5cdd81ae0e1626e4e3769220594</version>
    </contained_entity>
  </contained_entities>
</sofanet_distribution_package>
```

Listing 4.2: Example of distribution package `contents.xml` file

The distribution package contains many zip archives (entity package), but it is also a zip archive of its own. It can be understood as a two layer archive, each layer represents a different level of abstraction. An overview of the whole structure is presented in *figure 4.1: Tree Structure of the Distribution Package.*



```
logdemo_1_0.sdp
    contents.xml
    org.objectweb.dsrg.sofa.examples.logdemo.arch.LogDemo-574e0ec44fb2c1a09e246d829bd6cfd8c5b070a5.sp
        adl.xml
        contents.xml
    org.objectweb.dsrg.sofa.examples.logdemo.arch.Logger-da1a752b079a3d30a18b042ac3c20f4d018aa832.sp
        adl.xml
        contents.xml
        org.objectweb.dsrg.sofa.examples.logdemo.arch.Logger.jar
    org.objectweb.dsrg.sofa.examples.logdemo.arch.Tester-195475353337f1744401c3ed92c9bb01eb25cd83.sp
        adl.xml
        contents.xml
        org.objectweb.dsrg.sofa.examples.logdemo.arch.Tester.jar
    org.objectweb.dsrg.sofa.examples.logdemo.assm.LogDemo-4a87f0d8515fa5cdd81ae0e1626e4e3769220594.sp
        adl.xml
        contents.xml
    org.objectweb.dsrg.sofa.examples.logdemo.frame.LogDemo-9ad094b443cb8dbffca09424209c4c023f4436cf.sp
        adl.xml
        contents.xml
    org.objectweb.dsrg.sofa.examples.logdemo.frame.Logger-48f5669aa0d01d1e492bda81ca80a1fa6e1c0056.sp
        adl.xml
        contents.xml
    org.objectweb.dsrg.sofa.examples.logdemo.frame.Tester-3b9343ef66d34fe19d02a13b1364549e8169421f.sp
        adl.xml
        contents.xml
    org.objectweb.dsrg.sofa.examples.logdemo.iface.Log-24ec551ad1e590948c93a74a9f8f3698c1bb435d.sp
        adl.xml
        contents.xml
        org.objectweb.dsrg.sofa.examples.logdemo.iface.Log.jar
```

Figure 4.1: Tree Structure of the Distribution Package

5. Implementation

The main business logic of the SOFAnet 2 is implemented in a module called SOFA Tools API, which we describe in the following section. The description of the graphical and command line user interfaces of SOFAnet 2, which both use the shared logic in the SOFA Tools API, is presented in the second half of this chapter.

5.1 SOFA Tools API

5.1.1 Introduction to SOFA Tools API

SOFA Tools API (further just *API*) provides a set of functions covering essential development and deployment tasks. It is intended to contain the common application logic of all SOFA 2 tools. Today it is a stand-alone module shared among Cushion, SOFA 2 IDE and MConsole. Its placement in the architecture of SOFA 2 is illustrated in *figure 5.1: Placement of SOFA 2 Tools API in the SOFA 2 Framework*.

5.1.2 Concept of Actions

From the client's point of view, the *API* is a set of actions, each implementing one functionality. An *action* is a class with various `perform`[1] methods. It typically interacts with the repository and workspace[2]. The *actions* also provide some outputs such as errors, warnings and informational messages. For more information about the *API* and *actions*, please refer to [22].

5.1.3 Common Interface to SOFAnet 2 Actions

The *actions* that were previously implemented in the *API* were mostly operating on entities and were relatively quick. The SOFAnet 2 *actions* operate mostly on *packages* and can take extensive amounts of time. Therefore the typical interface of those *actions* is somewhat different. It usually operates with two concepts described in the following paragraphs.

DistributionPackageInfo Class

The `DistributionPackageInfo` class encapsulates information about a *package*, including:

- name

[1] An *action* is usually initialized in its constructor. The operations the *action* implements are performed inside its `perform` method, which should get all its inputs as parameters passed to it upon call. For more details see [22].

[2] A storage of entities, that are in the process of creation or are checked out for modifications.

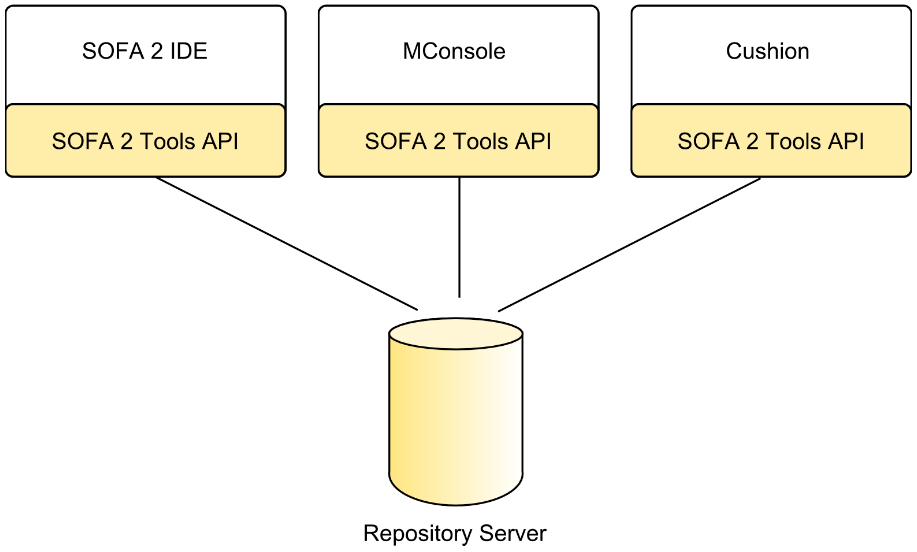


Figure 5.1: Placement of SOFA 2 Tools API in the SOFA 2 Framework
Originally published in [22].

- version
- type (e.g. single *Application*, single *Component*, multiple *Components*, etc.)
- main entities (*Assemblies* and *Architectures*)
- all the contained entities

It can provide the same information that is contained in the distribution package `contents.xml` file (see *section 4.4.2: Composition of Distribution Package* for more details). An instance of the `DistributionPackageInfo` class can be created from this file or from a collection of entities taken from the repository. In this case, the contained entities are computed on demand (using the requirements closure).

IProgressNotification Interface

The `IProgressNotification` interface was created to allow passing of progress information from *actions*, that are expected to run for a long time. It can also pass a request for early termination from a user interface (UI) to the action logic. An interface was chosen, because progress notification is desirable also in the form of Eclipse progress dialog[3], while the *API* should stay as platform-and-framework independent as possible. This way, the `IProgressNotification` can be implemented inside the client Eclipse application.

There is one default implementation of the `IProgressNotification`, that is used for convenience whenever no progress notification is required. It is called `NullProgressNotification` class, because it discards all progress information calls and

[3]SOFA 2 IDE is built upon Eclipse Framework.

always tells the *action* to continue its work.

Input and Output Stream Factories

Both the entity packages and the distribution packages are expected to be transferred by various means. For example, an entity package may be stored by itself on a disk or it may be stored inside a distribution package, which is then transferred over the Internet. Therefore, the concept of input and output stream factories was introduced to the interfaces of *actions*.

The `IInputStreamFactory` and `IOutputStreamFactory` are generic interfaces to create input and output streams. They are implemented by the `ZipInputStreamFactory`, `ZipOutputStreamFactory`, `FileInputStreamFactory` and `FileOutputStreamFactory` and allow chaining. The package transfer over the Internet is implemented as an inner class, that implements both `IInputStreamFactory` and `IOutputStreamFactory` using a byte array and is being chained with the zip stream factories.

5.1.4 Actions for Distribution

Export and Import

The `Export` and `Import` *actions* handle *entity packages*. The `Export` is for package creation, `Import` for installation into a repository.

CreatePackage and InstallPackage

Similarly to the way how `Export` and `Import` work with *entity packages*, the `CreatePackage` and `InstallPackage` *actions* handle *distribution packages*. Please refer to the *section 4.3.2: Semantics of the Off-line Application Transfer Use Case* for the description of their semantics.

TransferPackage

The `TransferPackage` *action* supports the *On-line Application Transfer use case*. Although its semantics was defined in *section 4.3.3: Semantics of the On-line Application Transfer Use Case* using a *create_package* function, it does not create a full fledged package. It transfers only the entities that have to be transferred, because it is desirable to spare time by lowering the amount of data to be transferred. It also eliminates unnecessary entity queries to both repositories by utilizing caches. The requirements of the entities, that are present in the target repository prior to the transfer, are not even queried from the source repository.

5.1.5 Actions for Unistallation

The actions that handle removal of entities are using `IRemovalConfirmation` interface. This interface should be implemented in the client of the *API* and let the user confirm the list of entities to be removed (or cancel the operation).

GarbageCollect

The `GarbageCollect` *action* is implemented according to the *section 4.3.6: Semantics of the Garbage Collection Use Case*. When a Node Administrator decides to refine the set of entities to be removed, the `MarkDevelopment` *action* (described in the *section 5.1.6: Supporting Actions*) should be utilized.

Uninstall

The `Uninstall` *action* supports the *Uninstallation use case*. It should be used together with the `MarkDevelopment` *action* the same way as the `GarbageCollect` *action*. The `GarbageCollect` *action* should be optionally triggered by the client of the *API* after a successful uninstallation.

5.1.6 Supporting Actions

The main *actions* described above are accompanied with the following auxiliary *actions*.

AvailablePackages

The `AvailablePackages` *action* examines a repository and returns a list of available component and/or application packages.

ExaminePackage

Use the `ExaminePackage` *action* to get a user friendly description of a *distribution package*. Both the packages created from a repository during runtime and package files are supported.

MarkDevelopment

The `MarkDevelopment` *action* can set, unset and query the *Development Marks* on entities (see the *definition 15* in *section 4.3.4: Making Order in a Repository*).

5.1.7 Set Operations with Entities

EntitiesFilter

In the previous chapter a model of the solution has been introduced. In the model, the repository and packages are modeled as sets. To implement the calculations with sets required by the model, a class called `EntitiesFilter` was created.

Given a repository, the class evaluates an expression with operators, parentheses, basic set constructors and functions. It returns a set of entities as a result. During the evaluation, internal entities are not taken into account, because they should not be manipulated.

We feel that set operations upon entities may be utilized for various purposes. The class was made extensible to maximize its chances of reuse.

Basic Set Constructors

Basic Sets act as terms for the expression. In order to create a *basic set*, the entities from a repository are filtered based on:

- name
- version
- type
- tag
- development marks

The names, versions and tags are matched using regular expressions. A constructor that creates a set containing only a given entity is supported too.

New *basic set* constructors can be easily added. Each *basic set* constructor implements a `IFilter` interface. New set constructors can be passed to the `EntitiesFilter` instance while calling its constructor.

Operators

Furthermore, these operators are supported on sets:

- union
- intersection
- complement

All the operators are build-in and can not be easily customized.

Functions

Functions were introduced to allow the computations of closures. All functions must implement the `IFunction` interface. Custom implementations can be passed to the `EntitiesFilter` instance while calling its constructor too. By default, two functions implementing the closures are recognized, namely:

- dependants
- requirements

Both of them take only one parameter, because they operate on the repository that was given to the `EntitiesFilter` instance upon its construction.

Example

In the the *definition 18* in the *section 4.3.4: Making Order in a Repository* we have defined a set K. In one of the cases, K can be computed using the following formula:

$$K = dependants(e) \cup deployment\_plans(R)$$

This formula can be evaluated by the `EntitiesFilter` class. First, the instance of the class needs to be created for the repository R. Then the following `String` can be passed as an expression for evaluation:

`dependants( entity=`$< e.name >$`:`$< e.version >$`) OR type=deploymentplan`

The result of the evaluation will be the desired value of the set K.

5.1.8 Addressing Consistency and Concurrency

Only One Interface for the Repository

The SOFAnet 2 implementation introduced no changes in the repository or its interface[4]. Therefore it accesses the repository in a common way with all the other tools. No inconsistencies should be introduced by sequential operations of various tools, no matter whether they are aware of SOFAnet 2 or not.

Locks on the Repository

The concurrent operation of the tools on a single repository is solved by non-mandatory locks, implemented on the repository server side. Those locks were honored by some of the more complex *actions* even previously. The SOFAnet 2 actions of course acquire a lock when needed. Therefore the operations that can influence the repository consistency or have important results[5] are forced to be running sequentially.

5.2 GUI

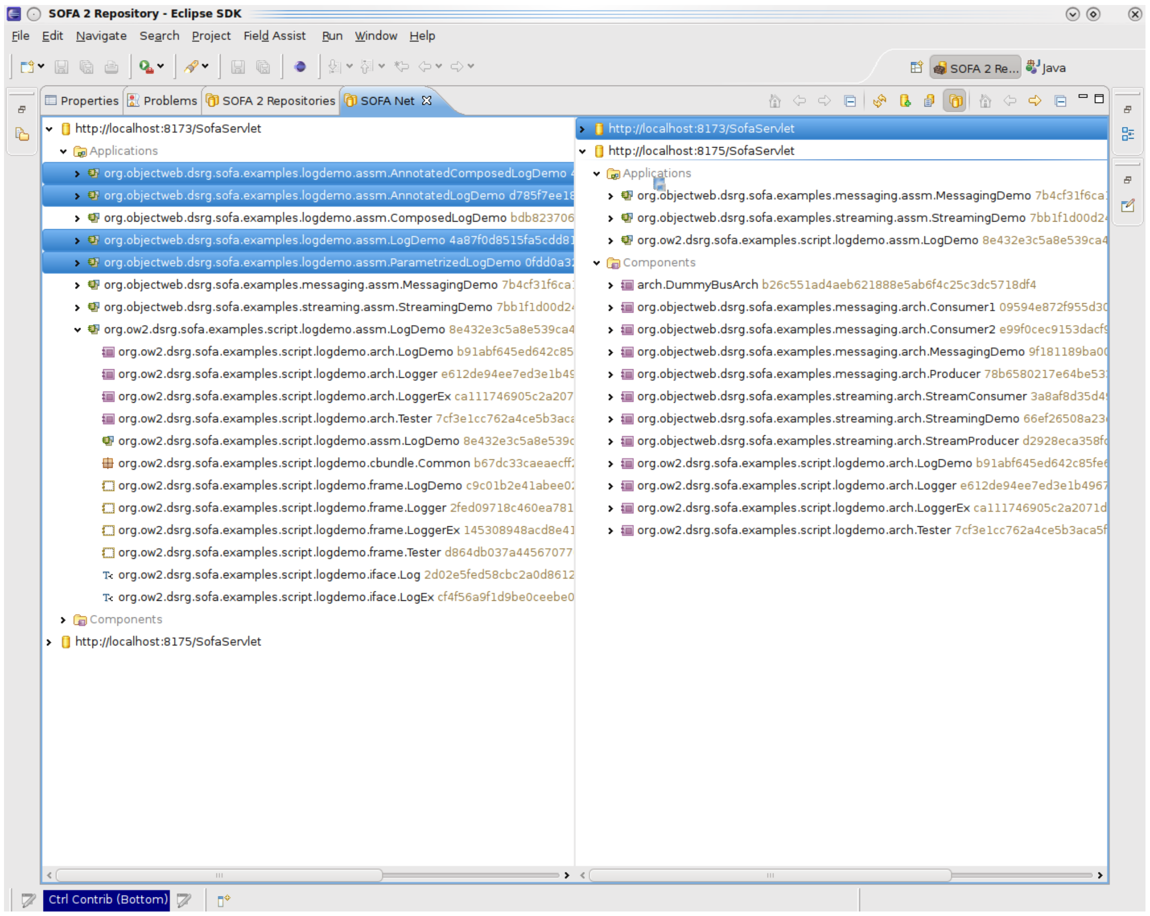
The Graphical User Interface (GUI) for the SOFAnet 2 was integrated into the SOFA 2 IDE.

5.2.1 Introduction to SOFA 2 IDE

The SOFA 2 IDE is a graphical environment for developing SOFA 2 applications. It is built as an Eclipse IDE [12] plug-in. Its architecture is well described in [22], therefore we will focus only on the integration of the SOFAnet 2 features.

[4]`RepositoryAgent` and `RepositoryFacade` classes.

[5]For example `CreatePackage` or `Uninstall` *actions* do lock the repository. On the other hand, `ExaminePackage` *action* does not, because it introduces no changes to the repository and its result is only informative.


SOFA 2 Repository - Eclipse SDK
File Edit Navigate Search Project Field Assist Run Window Help
SOFA 2 Re...
Java
Properties
Problems
SOFA 2 Repositories
SOFA Net
Ctrl Contrib (Bottom)

```
http://localhost:8173/SofaServlet
  Applications
    org.objectweb.dsrg.sofa.examples.logdemo.assm.AnnotatedComposedLogDemo
    org.objectweb.dsrg.sofa.examples.logdemo.assm.AnnotatedLogDemo d785f7ee18
    org.objectweb.dsrg.sofa.examples.logdemo.assm.ComposedLogDemo bdb823706
    org.objectweb.dsrg.sofa.examples.logdemo.assm.LogDemo 4a87f0d8515fa5cdd8
    org.objectweb.dsrg.sofa.examples.logdemo.assm.ParametrizedLogDemo 0fdd0a3
    org.objectweb.dsrg.sofa.examples.messaging.assm.MessagingDemo 7b4cf31f6ca
    org.objectweb.dsrg.sofa.examples.streaming.assm.StreamingDemo 7bb1f1d00d2
    org.ow2.dsrg.sofa.examples.script.logdemo.assm.LogDemo 8e432e3c5a8e539ca4
      org.ow2.dsrg.sofa.examples.script.logdemo.arch.LogDemo b91abf645ed642c85
      org.ow2.dsrg.sofa.examples.script.logdemo.arch.Logger e612de94ee7ed3e1b4
      org.ow2.dsrg.sofa.examples.script.logdemo.arch.LoggerEx ca111746905c2a207
      org.ow2.dsrg.sofa.examples.script.logdemo.arch.Tester 7cf3e1cc762a4ce5b3ac
      org.ow2.dsrg.sofa.examples.script.logdemo.assm.LogDemo 8e432e3c5a8e539
      org.ow2.dsrg.sofa.examples.script.logdemo.cbundle.Common b67dc33caeaecff
      org.ow2.dsrg.sofa.examples.script.logdemo.frame.LogDemo c9c01b2e41abee0
      org.ow2.dsrg.sofa.examples.script.logdemo.frame.Logger 2fed09718c460ea781
      org.ow2.dsrg.sofa.examples.script.logdemo.frame.LoggerEx 145308948acd8e41
      org.ow2.dsrg.sofa.examples.script.logdemo.frame.Tester d864db037a44567077
      org.ow2.dsrg.sofa.examples.script.logdemo.iface.Log 2d02e5fed58cbc2a0d8612
      org.ow2.dsrg.sofa.examples.script.logdemo.iface.LogEx cf4f56a9f1d9be0ceebe0
  Components
http://localhost:8175/SofaServlet
```

```
http://localhost:8173/SofaServlet
http://localhost:8175/SofaServlet
  Applications
    org.objectweb.dsrg.sofa.examples.messaging.assm.MessagingDemo 7b4cf31f6ca
    org.objectweb.dsrg.sofa.examples.streaming.assm.StreamingDemo 7bb1f1d00d2
    org.ow2.dsrg.sofa.examples.script.logdemo.assm.LogDemo 8e432e3c5a8e539ca4
  Components
    arch.DummyBusArch b26c551ad4aeb621888e5ab6f4c25c3dc5718df4
    org.objectweb.dsrg.sofa.examples.messaging.arch.Consumer1 09594e872f955d3
    org.objectweb.dsrg.sofa.examples.messaging.arch.Consumer2 e99f0cec9153dacf
    org.objectweb.dsrg.sofa.examples.messaging.arch.MessagingDemo 9f181189ba0
    org.objectweb.dsrg.sofa.examples.messaging.arch.Producer 78b6580217e64be53
    org.objectweb.dsrg.sofa.examples.streaming.arch.StreamConsumer 3a8af8d35d4
    org.objectweb.dsrg.sofa.examples.streaming.arch.StreamingDemo 66ef26508a23
    org.objectweb.dsrg.sofa.examples.streaming.arch.StreamProducer d2928eca358f
    org.ow2.dsrg.sofa.examples.script.logdemo.arch.LogDemo b91abf645ed642c85fe
    org.ow2.dsrg.sofa.examples.script.logdemo.arch.Logger e612de94ee7ed3e1b4967
    org.ow2.dsrg.sofa.examples.script.logdemo.arch.LoggerEx ca111746905c2a2071d
    org.ow2.dsrg.sofa.examples.script.logdemo.arch.Tester 7cf3e1cc762a4ce5b3aca5f
```

Figure 5.2: SOFA Net View in Two Panel Mode

5.2.2 SOFA Net View

A new view called the SOFA Net View was introduced into the SOFA 2 Repository Perspective.

Single and Two Panel Mode

The view can be set to work in Single or Two Panel Modes (see *figure 5.2: SOFA Net View in Two Panel Mode* for example). The Single Panel Mode is convenient when a user wants to see whole entity names and versions, because both of them are quite long. The Two Panel Mode is intended to be used when transferring packages between repositories.

Drag and Drop

The `TransferPackage` *action* is invoked when a selection of Applications and Components is dragged and dropped on a SOFAnode location. The selection can be dropped into both panels no matter where the selection was chosen from.

Context Menu Actions

There are multiple actions available through the context menu. On the SOFAnode Location, the `InstallPackage` action is enabled.

For selections of Applications and Components, the `CreatePackage` and `ExaminePackage` actions are available.

Toolbar Extensions

The contents of the SOFA Net View toolbar is changing with the view mode. There are three groups of buttons in the Two Panel Mode and two groups of buttons in the Single Panel Mode. The missing group in the Single Panel Mode is the tree navigation button group for the second panel.

There are two important buttons in the action group in the middle. One is for switching the panel modes of the view. The other is invoking the `ExaminePackage` action, that examines a *package file*.

5.2.3 Extensions to the SOFA Repository View

The uninstallation and garbage collection functions were integrated in the SOFA Repository View.

Its label provider was updated to display the status of the *Development Marks*.

Context Menu Actions

The other actions were added to the context menu as follows. On the SOFAnode Location, the `GarbageCollect` and `Import` actions are available.

When a single entity is selected, the `Uninstall` and `MarkDevelopment` actions can be invoked. For one or more selected entities, the `Export` action is enabled.

5.3 CLI

The Command Line Interface (CLI) for the SOFAnet 2 is a part of the development and management command line tool for SOFA 2 called Cushion.

5.3.1 Introduction to Cushion

Cushion is a thin shell wrapping the SOFA Tools API. It uses the concept of *actions* too.

The main purpose of the Cushion *actions* is to parse the textual form of the parameters, convert them to the form the *API* accepts and delegate the call the corresponding action in *API*. Therefore the *actions* in Cushion usually directly correspond to the *actions* in *API*.

Please refer to [22] for more details.

5.3.2 CommonPackagingCLI Class

In Cushion, each *action* should parse its own command line parameters and also should provide the user with help. However, the parameters of the SOFAnet 2 actions mostly overlap. Therefore the `CommonPackagingCLI` class was introduced to hold all the common parameters, their help strings and handle their parsing. This way, the CLI for all the SOFAnet 2 actions is consistent and easy to modify.

5.3.3 SOFAnet 2 Actions in Cushion

In case of the SOFAnet 2, the Cushion actions exactly correspond to the actions in the *API*. These were described in *section 5.1.4: Actions for Distribution*, *section 5.1.5: Actions for Unistallation* and *section 5.1.6: Supporting Actions*.

6. Evaluation and Related Work

6.1 Evaluation

This section presents the evaluation of our SOFAnet 2 prototype implementation, that is on the enclosed DVD (see *appendix B: Contents of the Enclosed DVD* for details). The latest version is available for download from [25].

6.1.1 The Main Use Cases

The implementation was tested to support the main use cases presented in *section 3.1: Use Cases in Focus* in the first place. During the development, small examples called LogDemos, that are part of SOFA 2 distribution were used. For final testing, SOFA SHOP, that is also available in the SVN repository of the SOFA 2 [25], was utilized as an example of larger and more complex application.

The implementation proved to handle all the use cases well. The GUI is more comfortable because it provides better overview and displays in advance information about what exactly will be done. On the other hand, the CLI displays information only when user asks for it. The CLI also has better mechanisms to choose the concepts we want to operate with. Therefore, it does not query unnecessary data and saves a considerable amount of network traffic.

Both interfaces work fast enough, although a progress feedback is necessary. For example, the on-line transfer of an Application will take from tens of seconds to a few minutes depending on its size. The operation can be performed approximately 20 % to 30 % faster by the CLI, because the CLI does not pre-compute what should be transferred only to show the list to the user in advance.

6.1.2 Use Cases Not In Our Main Focus

Our solution is focused on the high level concepts of *Applications* and *Components*. Sometimes, although it is not our main concern, someone may want to distribute some software parts with finer granularity. For example, someone may want to distribute a new *Aspect* or a *Frame* with *Interface Types*. How our prototype implementation handle this case?

Even such use cases are supported, but in a bit more complicated way. In case of the off-line transfer, entity packages can be created by either SOFA 2 IDE or Cushion[1]. These tools allow exporting a root entity with all its requirements into a directory. The user that has to pack those entity packages manually into some

[1]Utilizing `Export` action. The SOFA 2 IDE does not support entities from the micro-architecture meta-model. Cushion supports all entities except *Deployment Plans*.

kind of archive. The receiver of such an archive needs to unpack it and then use our system to install all the entities[2].

For on-line transfer the `RepositoryCloner`[3] class can be abused. This class is intended to clone runtime form of entities, so it is the most fragile tool if the versions of SOFA 2 on both sides are not perfectly matching. On the other hand, this is now the only tool that can transfer even *Deployment Plans*.

The unistallation was proposed in such a way, that it works not only with *Applications* and *Components*, it can uninstall any entity in general. However, we are not sure whether it makes a good sense. It is accompanied with garbage collection that works on entity level.

To sum up, even though the fine granularity support was not our main concern, it is possible to achieve with the current prototype implementation.

6.2 Related Work

SOFA 2 is not the only component system of its kind and the SOFAnet 2 is certainly not the only solution that covers software distribution in scope of component systems. A representative selection of projects that are either component systems or target component distribution follows. Their capabilities compared to SOFAnet 2 are discussed shortly.

6.2.1 COM/DCOM, EJB, CCM and Fractal

We have mentioned the Microsoft's Component Object Model (COM) and Distributed Component Object Model (DCOM) [1], Enterprise JavaBeans (EJB) [3], CORBA Component Model (CCM) [4] and Fractal [9] in *chapter 1: Introduction*. Those component systems correspond to the SOFA 2. Some are more similar such as Fractal, some less, like COM, which is closed proprietary technology without support for distributed computing. These component systems either do not target software distribution at all or it plays a marginal role for them.

EJB distributes and deploys applications in JAR [13] files. The JAR files are a product of application compilation, therefore only the developers or especially trained administrators know precisely about their dependencies and are able to transfer larger applications from one corporate environment into another.

The CCM specification deals with packaging. In defines Component, Assembly and Software Descriptors. Furthermore packages in form of zip archives consisting of one Software or Assembly Descriptor and files containing implementation of components can be composed. The Descriptors contain additional information such as name, author, target platform or dependencies and are implemented as a xml file. All

[2] `Import` action should be used for that purpose. It can install an entity with all its requirements from a given directory. However, the user needs to know which package contains the root entity.

[3] Contained in the `org.objectweb.drsg.sofa.repository` package.

the implementations of components are uniquely identified by the Universal Unique Identifier (UUID), which is used in COM and DCOM too.

6.2.2 OSGi Service Platform

OSGi Service Platform [26] is a standard describing dynamic module system for Java. Its specification is based on bundles. Both applications and components can come in a form of a bundle, for which a full life cycle is defined. The bundles can be installed, started, stopped, updated and uninstalled without requiring a reboot, and all of this can be done using remote access. They can cooperate using services, which they can look up in a service registry. Therefore they are able to adapt to the addition or removal of services during runtime.

Although the OSGi was focused on embedded systems like home electronics at the time of its introduction, today it is being used even for enterprise applications, because it is one of the most advanced module systems for Java. For example, the main unit of distribution and deployment in the Spring Framework [27] is an OSGi bundle and Eclipse IDE uses OSGi as its runtime and plug-in module system too.

There are several certified implementations of OSGi [26]. For example Knopflerfish [28] and Equinox [29], and many other without certification like for example Oscar [30]. However, the means of bundle transfer are left by the standard on implementation. The vendors have created Bundle Repositories and support only a pull model. Many OSGi Bundle Repositories that allow automatic bundle installation with a transitive closure of bundle's dependencies are available.

6.2.3 Universal Packaging

There are many formats of packages for software distribution, that are usually close-knit to a particular operating system or platform.

From the platform related, a representative example is a JAR file [13] used in the Java 2 Platform [31]. It is a zip archive that contains implementations of classes and meta-data information in so called manifest file. The meta-data from the manifest file is not used by the packaging system itself (e.g. for package installation), therefore this kind of package is one of the simplest we can find.

From the many package formats tightly bound to operating systems, the formats of the RPM Package Manager [32] and Debian Packages [33] are representative. Both of them are used in various GNU/Linux [34] [35] distributions. They are more advanced in comparison to JAR files, because they contain information about configuration, dependencies and installation, that the packaging system actually utilizes. We think that the MSI format of the Windows Installer [36] fits in this category too, because it is a bare data file.

The last approach is utilizing directly executable packages. Such packages include complete package management system inside. They are common for applications running on Microsoft Windows [37] operating systems. Some applications are distributed in the form of Shell scripts for GNU/Linux platforms, like for example NetBeans IDE [38].

6.2.4 Update Services

Both the users and the vendors usually want to keep the installed software up-to-date with the least possible effort for the users. Update services are based on a pull model of software distribution. The service on the user's computer contacts the providers update repository and searches available updates. Not only for security reasons, the users are usually asked if they want to download and install the updates found.

The operation systems equipped with their package management system, like the RPM or Debian based, usually utilize an integrated update service.

Commercial software providers employ proprietary update services, that usually take care of all the provider's product installed on a user's computer. We can mention for example Windows Update [39], Adobe Updater [40] or Omaha [41], also known as Google Update. Some programs occasionally have simple update systems built into them.

7. Conclusion and Future Work

7.1 Summary

This thesis builds up on a rich plateaux of previous work on the original SOFAnet. Our goal was to recreate the SOFAnet for SOFA 2 emphasizing reasonable distribution and fundamental use cases.

A detailed analysis was done. New high level concepts were introduced to reasonably reflect our main use cases. A mapping of the new concepts to the SOFA 2 concepts was presented. Furthermore, a methodology to keep the repository clean was worked out. The proposed solution was mathematically modeled and tested upon a prototype implementation, which has proven to fulfill our requirements.

On the other hand, the thesis has been focused only on the distribution and fundamental use cases. Many features of the former SOFAnet have not been discussed nor implemented. Of the most important ones we can mention licensing and the search and share networks.

The main goals of the thesis were achieved.

7.2 Contribution

The thesis has contributed to the SOFA 2 project in many areas.

We have worked out a new high level abstraction, namely *Applications* and *Components*, and provided its mapping to the lower abstraction level (SOFA 2 entities).

We have also implemented a way of reasonable component and application exchange between SOFA 2 repositories, both using a direct connection over the Internet and file based.

SOFAnet 2 is also a way of keeping order in the repository. Methodology was proposed and functions for *Application* uninstallation and garbage collection were implemented.

7.3 Comparison to Previous Work

On first glance, SOFAnet 2 seams to be simpler in comparison to the original SOFAnet. We have not covered all the scope of SOFAnet, because our goals were somewhat different and more focused. Moreover, we have worked out deeper all the topics we were concerned about.

We were interested in the fundamental use cases around high level content, that were not sufficiently covered. Our way of distributing software parts in file based packages is new to SOFA 2 (and SOFA too). Also, the new SOFAnet 2 has been concerned about uninstallation and keeping the repository clean. Making order in the repository is a brand new and desired feature. The uninstallation was mentioned in the previous implementation of SOFAnet, but it was not aware of dependencies between entities. Therefore it was barely usable from our point of view.

On the other hand, many features of the original SOFAnet were not discussed at all, like licensing, triggers for automatic software distribution, and bundle searching and sharing in corporate networks. It was not possible in scope of a master thesis to cover all those topis and provide a reasonable and usable solution for the fundamental use cases at the same time. Therefore, we leave those topics for future work.

7.4 Future Work

There are many areas left for future work. As we have stated in the paragraph above, we have dropped many features of the original SOFAnet from the thesis. It was either because of the amount of work required or since they were not the current concern of our users.

From the field we were focused on we think that easy distribution of software to many receivers at once would bring a good value to the users. This can be implemented as a stand-alone distribution server from which the receivers can pull the software. Such a server could also allow access control and monitoring of the distribution in form of statistics and reports. To allow pushing software to multiple receivers at once, the current tools can be easily extended.

Many more features similar to the one mentioned above, that are desired but not implemented in the prototype implementation, can be found in *appendix A: Catalogue of Requirements.*

Bibliography

[1] Microsoft Corporation, *Component Object Model (COM)*, [Online]. Available: http://www.microsoft.com/com/

[2] Microsoft Corporation, *.NET Framework*, [Online]. Available: http://www.microsoft.com/net/

[3] Oracle Corporation, *Enterprise JavaBeans*, [Online]. Available: http://www.oracle.com/technetwork/java/javaee/ejb/

[4] Object Management Group, *CORBA Component Model*, [Online]. Available: http://www.omg.org/technology/documents/formal/components.htm

[5] Borland Software Corporation, *VisiBroker*, [Online]. Available: http://techpubs.borland.com/am/visibroker/v80/

[6] *SOFA Component System*, [Online]. Available: http://sofa.ow2.org/sofa1/index.html

[7] Hnětynka, P., Píše, M.: *Hand-written vs. MOF-based Metadata Repositories: The SOFA Experience*, Proceedings of ECBS 2004, Brno, Czech Republic, IEEE CS, May 2004.

[8] *SOFA 2 Component System*, [Online]. Available: http://sofa.ow2.org/

[9] *Fractal Component System*, [Online]. Available: http://fractal.ow2.org/

[10] Object Management Group, *MetaObject Facility*, [Online]. Available: http://www.omg.org/mof/

[11] *AspectJ*, [Online]. Available: http://www.eclipse.org/aspectj/

[12] The Eclipse Foundation, *Eclipse IDE*, [Online]. Available: http://www.eclipse.org/

[13] Oracle Corporation, *JAR File Specification*, [Online]. Available: http://download.oracle.com/javase/6/docs/technotes/guides/jar/jar.html

[14] Bureš, T., Hnětynka, P., Plášil, F.: *SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model*, Proceedings of SERA 2006, Seattle, USA, IEEE CS, ISBN 0-7695-2656-X, pp. 40-48, Aug 2006.

[15] Bureš, T., Hnětynka, P., Plášil, F.: *Runtime Concepts of Hierarchical Software Components*, In International Journal of Computer & Information Science, Vol. 8, No. S, ISSN 1525-9293, pp. 454-463, Sep 2007.

[16] Bureš, T., Mencl, V.: *Microcomponent-Based Component Controllers: A Foundation for Component Aspects*, in Proceedings of 12th Asia-Pacific Software Engineering Conference (APSEC 2005), Dec 15-17, 2005, Taipei, Taiwan, pp. 729-738, ISBN 0-7695-2465-6, ISSN 1530-1362, IEEE Computer Society Press, Dec 2005.

[17] Plášil, F., Bálek, D. and Janeček, R.: *SOFA/DCUP: Architecture for Component Trading and Dynamic Updating*, Proceedings of ICCDS'98, Annapolis, Maryland, USA, IEEE CS Press, May 1998.

[18] Panuška, P.: *An Approach to SW Distribution*, Reviewed section of Proceedings of the Week of Doctoral Students 2003 conference (WDS 2003), pp.118-123, Matfyzpress, Charles University in Prague, Prague, Czech Republic, June 2003.

[19] Šobr, L.: *Network Environment of the SOFA Architecture*, Master thesis, Charles University in Prague, 2004.

[20] Šobr, L., Tůma, P.: *SOFAnet: Middleware for Software Distribution over Internet*, Proceedings of the 2005 Symposium on Applications and the Internet (SAINT'05), Washington, DC, USA, IEEE Computer Society, ISBN 0-7695-2262-9, pp.48-53, 2005.

[21] *SOFA 2 Component System User's Guide*, [Online]. Available: `http://sofa.ow2.org/docs/`

[22] *SOFA 2 Component System Developers's Guide*, [Online]. Available: `http://sofa.ow2.org/docs/`

[23] Hnětynka, P., Plášil, F., Bureš, T., Mencl, V., Kapová, L.: *SOFA 2.0 metamodel*, Charles University in Prague, Tech. Rep. 2005/11, Dec 2005.

[24] Keznikl, J., *SOFA 2 runtime support for dynamic languages*, Master thesis, Charles University in Prague, 2010.

[25] *SOFA 2 Project SVN Repository*, [Online]. Available: `svn://svn.forge.objectweb.org/svnroot/sofa`

[26] OSGi Alliance, *OSGi Service Platform*, [Online]. Available: `http://www.osgi.org`

[27] SpringSource, *Spring Framework*, [Online]. Available: `http://www.springsource.org/`

[28] Makewave, *Knopflerfish Pro 3*, [Online]. Available: `http://www.makewave.com/`

[29] The Eclipse Foundation, *Equinox 3.2*, [Online]. Available: `http://www.eclipse.org/equinox/`

[30] Hall, R. S., *Oscar*, [Online]. Available: http://oscar.ow2.org/index.html

[31] Oracle Corporation, *Java 2 Standard Edition*, [Online]. Available: http://download.oracle.com/javase/

[32] *RPM Package Manager*, [Online]. Available: http://www.rpm.org/

[33] Software in the Public Interest, Inc., *Debian Packages*, [Online]. Available: http://www.debian.org/distrib/packages

[34] Free Software Foundation, Inc., *GNU Operating System*, [Online]. Available: http://www.gnu.org/

[35] *Linux Kernel*, [Online]. Available: http://www.kernel.org/

[36] Microsoft Corporation, *Windows Installer*, [Online]. Available: http://msdn.microsoft.com/en-us/library/cc185688(VS.85).aspx

[37] Microsoft Corporation, *Windows*, [Online]. Available: http://www.microsoft.com/windows/

[38] Oracle Corporation, *NetBeans IDE*, [Online]. Available: http://netbeans.org/

[39] Microsoft Corporation, *Windows Update*, [Online]. Available: http://update.microsoft.com

[40] Adobe Systems Incorporated, *Adobe Updater*, [Online]. Available: http://www.adobe.com/support/downloads/product.jsp?product=165&platform=Macintosh

[41] Google Incorporated, *Omaha*, [Online]. Available: http://code.google.com/p/omaha/

A. Catalogue of Requirements

This catalogue further elaborates the use cases from *section 3.1: Use Cases in Focus* and adds requirements desired for any package distribution and management system. For a description of Stake Holders please refer to *section 3.1.1: Stake Holders*.

A.1 Data dictionary

The following table summarizes and describes the terms that were used without definition in the use cases. It also brings up some new terms that were needed to be introduced in the requirements.

name	description
package	The basic unit that the system will use to distribute applications, their parts or application updates.
update package	Special kind of a *package*, that brings updates and that is dependent on the preceding base version *package*.
entity	*SOFA2 entity*, the smallest complete part of an application, that can be stored in *SOFA2 repository* and operated by *SOFA2* tools.
repository	*SOFA2 repository*; it stores all information about application components (in form of *entities*) as data and also some metadata.
package repository	Stores packages that were transferred to the local *SOFAnode*. It is accessible only locally and only for the *node administrator*. See requirement AM1 for details.
distribution server	Place where packages are accessible by *node administrators* over the Internet. This might by realized for example as a running server instance that is waiting for the connections. See requirement *D5* for details.

A.2 Priorities

When planning work on a system of significant size, it is useful to assign priorities to the tasks in the catalogue separately. We decided to denote the priorities in the following manner:

abbrev	name	description
E	Essential	Cannot be done without.
H	High value	Can be done without, although it may be very undesirable to do so.
L	Follow on	It is not clear whether they should be included in the first release.
O	Count with	Not included in the first release, but are desirable in the future.
X	Exempt	Features that are not going to be implemented.

A.3 Overview

We have extracted the requirements from the use cases (*section 3.1: Use Cases in Focus*) and added other requirements that are usually desired for any package management system. Because they are numerous, the tables below are presenting an overview, detailed description follows in formatted paragraphs.

A.3.1 Package creation

Defining new packages

abbrev	priority	short description
CN1	E	Guidelines and tool(s) to generate *packages*
CN2	E	Specify which *entities* should be included in a *package*
CN3	L	Specify dependencies on other *packages*
CN4	L	Decide which *entity* to pack and which to obtain elsewhere
CN5	L	Find out easily the *packages* on which to depend
CN6	X	Specify files to be delivered with a *package*
CN7	X	Specify dependencies on files

Defining update packages

abbrev	priority	short description
CU1	H	Deliver updates in form of complete/whole *packages*
CU2	L	Deliver updates in sparse packages
CU3	L	Deliver cumulative updates
CU4	X	Deliver change at the file level

Tools behavior

abbrev	priority	short description
CT1	E	Manual creation using interactive CLI
CT2	H	Manual creation using script
CT3	L	Manual creation using GUI
CT4	X	Automatic *package* creation for new application versions

A.3.2 Package administration

Monitoring

abbrev	priority	short description
AM1	X	Manually obtain and submit *packages*
AM2	E	List content and meta-data of a *package*
AM3	H	Search local *packages* by meta-data
AM4	H	Search on known *distribution servers* on request
AM5	L	Search on known *distribution servers* on request by meta-data
AM6	L	Automatically check *distribution servers* for updates
AM7	X	Browse *packages* by groups

Installation

abbrev	priority	short description
AI1	E	Install a *package*
AI2	E	Install an *update package*
AI3	H	Download required third party bundles
AI4	L	Install specific updates
AI5	L	Automatic update download and installation
AI6	X	Partial installation of a *package*
AI7	X	Manage multiple versions of a similarly named *package*
AI8	X	Preserve customizations to files across updates

Removal

abbrev	priority	short description
AR1	L	Uninstall a *package*
AR2	L	Uninstall an *update package*
AR3	X	Remove *package* from local *package repository*
AR4	X	Uninstall a *package* without source
AR5	X	Uninstall an *update package* without source
AR6	L	Remove not used *entities* after update
AR7	X	Remove not used files after update

A.3.3 Package transfer and distribution

abbrev	priority	short description
D1	E	Create an off-line distributable file
D2	H	Deliver *packages* in a compressed format
D3	H	Direct connection to both sides
D4	L	Direct connection with one supplier and multiple receivers
D5	L	Post *packages* and let the *node administrators* pull them
D5.1	L.E	Authorized access
D5.2	L.H	Limit access to contents by *node administrator*
D5.3	L.H	Show *package* meta-data to *node administrator*
D5.4	L.H	Show *package* content and meta-data to *distribution manager*
D5.5	L.L	Public access
D5.6	L.L	Report number of downloads by a *node administrator*
D5.7	L.O	Report results of the installation to *distribution manager*
D5.8	L.O	Send email notifications to *node administrators*/companies
D6	X	Licensing and license enforcement

A.3.4 Non-functional

abbrev	priority	short description
NF1	E	Complete command line functionality
NF2	L	GUI
NF2.1	L.E	GUI works with *SOFA 2 IDE*
NF2.2	L.E	GUI provides *package* creation and distribution through GUI
NF2.3	L.L	*Package* installation, monitoring and uninstallation
NF2.4	L.H	GUI works with *SOFA 2 MConsole*
NF3	H	Minimize UIs required for *package* management functions
NF4	H	High quality error messages
NF5	H	Relevant information should be logged to a log file
NF6	H	*SOFA 2 autoconfiguration* as default.
NF7	L	Sufficient busy/progress feedback

A.4 Requirements Description

This section presents in detail the requirements gathered in an early stage of the analysis. We have worked out the use cases that are desirable for any package management system, therefore the catalogue we made is relatively large. It was clear that we need to focus our work on a subset of the requirements. Certainly the subset that covers the use cases that are now the most important for users (see *section 3.1: Use Cases in Focus*) was chosen.

To put our first thoughts of all the desired features in contrast to the prototype implementation, we specify the status of requirement support after its description.

We want to emphasize that we present the requirements as they were made in the early analysis stage, only their current support statuses was added.

A.4.1 Package creation

Guidelines and tool(s) to generate *packages*

Abbreviation:	CN1
Priority:	Essential
Description:	Tool(s) that support the creation of a *package* must be implemented. The tool(s) must guide the *assembler* through all the steps of package creation and therefore should be preferably interactive.
Status:	Done.

Specify which *entities* should be included in a *package*

Abbreviation:	CN2
Priority:	Essential
Description:	The system should allow the *assembler* to choose the *entities* that should be inside the *package*. It should be able to include all dependencies of a chosen *entity* or select and deselect *entities* using filters. The filters should allow to choose the *entities* for example depending on: • name • applications they are part of • whether they are developed in-house or not • distributor • author • license • source • *packages* this *entity* is part of • commit date and should also allow wildcards.
Status:	The tools only support *packages* created from *Applications* and *Components*. These can be chosen by name, version and tag. All dependencies are always included, because they are integral parts of those concepts.

Specify dependencies on other *packages*

Abbreviation:	CN3
Priority:	Follow on
Description:	The system should allow the *assembler* to set automatically the dependencies on other *packages*. The *assembler* should also have the possibility to edit the list of generated dependencies manually or to construct the list from scratch.
Status:	Not supported. The tools only support *packages* created from *Applications* and *Components*. All dependencies are always included, because they are integral parts of those concepts. Finer granularity is not our main concern now.

Decide which *entity* to pack and which to obtain elsewhere

Abbreviation:	CN4
Priority:	Follow on
Description:	The system should allow the *assembler* to decide whether the required *entity* or *entities* (according to some filter) or whole required *packages* are packed or whether they are left as dependencies for the *node administrator* to resolve (this task might be supported also).
Status:	Irrelevant, because no dependencies between packages can arise. All required entities are integral parts of the concepts a package is composed of.

Find out easily the *packages* on which to depend

Abbreviation:	CN5
Priority:	Follow on
Description:	The system should inform the *assembler* in which *packages* the required *entities* belong.
Status:	Irrelevant, because no dependencies between packages can arise. All required entities are integral parts of the concepts a package is composed of.

Specify files to be delivered with a *package*

Abbreviation:	CN6
Priority:	Exempt
Description:	The system should be able to deliver files in the *packages*. That might be for example scripts for database creation, tools for configuring the delivered application or just data files.
Reason:	It's not clear how files fit in *SOFA 2* ideas. The meta-model does not handle them.

Specify dependencies on files

Abbreviation:	CN7
Priority:	Exempt
Description:	It should be possible to state that the *package* is dependent on a file (database, native library, configuration, ...), that is not necessarily delivered by the system in any *package*.
Reason:	It's not clear how files fit in *SOFA 2* ideas. The meta-model does not handle them.

Deliver updates in form of complete/whole ***packages***

Abbreviation:	CU1
Priority:	High value
Description:	The system should allow to deliver updates in a such form, that allow either installation over the previous version of a *package* or an installation from scratch.
Status:	Done.

Deliver updates in sparse packages

Abbreviation:	CU2
Priority:	Follow on
Description:	The system should allow to deliver a minimum-sized update, that would contain only the *entities* updated since the previous version of a *package* and that are required to run the application/library.
Status:	Not implemented. All required entities are integral parts of the concepts a *package* is composed of. Therefore this is not possible with the current package implementation.

Deliver cumulative updates

Abbreviation:	CU3
Priority:	Follow on
Description:	The system should be able to merge several sparse packages (described in *CU2*) into one cumulative package.
Status:	Not implemented. All required entities are integral parts of the concepts a *package* is composed from. Therefore this is not possible with the current package implementation.

Deliver change at the file level

Abbreviation:	CU4
Priority:	Exempt
Description:	The *update packages* should contain new versions of files, if the file has changed since previous version of application/library.
Reason:	It's not clear how files fit in *SOFA 2* ideas. The meta-model does not handle them.

Manual creation using interactive CLI

Abbreviation:	CT1
Priority:	Essential
Description:	The system should have a command line tool, that allows the *assembler* to interactively create a *package*.
Status:	Done.

Manual creation using script

Abbreviation:	CT2
Priority:	High value
Description:	The system should have a command line tool, that reads a script file and creates a *package* described in the script. The script format should be universal enough to help the *assembler* create various types of *package*, mainly the update ones. The script format should also support the creation of successive *update packages* without changing the script. Therefore, the script should for example do without exact versions or *entities*' names.
Status:	Partially done. Cushion supports scripts. The features related to the *update packages* were not implemented at all.

Manual creation using GUI

Abbreviation:	CT3
Priority:	Follow on
Description:	The system should have a tool with graphical user interface, that allows the *assembler* to create a *package* interactively .
Status:	Done.

Automatic *package* creation for new application versions

Abbreviation:	CT4
Priority:	Exempt
Description:	When a new version of an application or a library is finished (eg. new *Deployment Plan* is committed in the stable repository), the system should automatically create an *update package*, which might optionally be for disposal using the distribution mechanism the system.
Reason:	It is not clear how this should exactly work and what should trigger the automatic creation. The value of such a functionality for users is probably low.

A.4.2 Package administration

Manually obtain and submit *packages*

Abbreviation:	AM1
Priority:	Exempt
Description:	When the *node administrator* receives a *package* in a file, they should have an option to submit the *package* to the local *package repository*. Here the *package* should be available for search in sense of for example requirement *AM3*, but also all other *package* installation and removal functions should work upon the *packages* stored in a local *package repository*.
Reason:	Dropped because of simplicity and consistency reasons. It can introduce consistency problems when some of the SOFA 2 tools are not aware of it. All persistent data should be included preferably only in the repository.

List content and meta-data of a *package*

Abbreviation:	AM2
Priority:	Essential
Description:	The *node administrator* should have an option to list all the *entities* in a *package*, and all the meta-data, that are stored on the package level (not by individual *entities* in the *package*). This function should be available not only for all *packages* stored locally in a *package repository* (accessed by package name), but also for individual *packages* in form of files stored in various places on a file system (accessed by complete path).
Status:	Done.

Search local *packages* by meta-data

Abbreviation:	AM3
Priority:	High value
Description:	The *node administrator* should be provided with a function for searching all the *packages* stored in the local *package repository.* The system should be able to list all the *packages* by various filters on meta-data. The filters should allow to choose the *packages* for example depending on: • name • applications they are part of • whether they are created in-house or not • distributor • author • source • license • license-type • creation date • *entities* included • dependent *packages* or *entities* • required *packages* or *entities* • excluded *packages* or *entities* (conflicts) and should also allow wildcards.
Status:	Partially supported. Package resources are searched in a *repository* by name, type, version or tag.

Search on known *distribution servers* on request

Abbreviation:	AM4
Priority:	High value
Description:	Search for availability of *packages* by their name or search for package updates by name on all known *distribution servers* on request.
Status:	No proper support. Only one *repository* can be searched at a time. The tools do not distinguish between regular *packages* and *update packages* yet.

Search on known *distribution servers* on request by meta-data

Abbreviation:	AM5
Priority:	Follow on
Description:	Search for availability of *packages* or package updates on all known *distribution servers* on request. Search should be provided by the meta-data of the *package*, as described in requirement *AM3*. Extends requirement *AM4*.
Status:	No proper support. Only one *repository* can be searched at a time.

Automatically check *distribution servers* for updates

Abbreviation:	AM6
Priority:	Follow on
Description:	When any of the system tools starts, it should check whether there is a new update on any of the known *distribution servers*. This functionality should be optional, so that the *node administrator* tells the system for example in a configuration file, that it should not automatically query the *distribution servers*.
Status:	No support. A list of *distribution servers* or *repositories* to query is not implemented nor maintained.

Browse *packages* by groups

Abbreviation:	AM7
Priority:	Exempt
Description:	When searching or listing the *packages*, the tools should be able to sort of filter the *packages* according to the type or target field of the application or library (for example network management, software development, office, core system functions, etc.).
Reason:	It is too complicated compared to a relatively low value. The *assembler* would have to specify the target field of the application or the *package*. This might end up so, that all the *packages* are in one group called “general” or “unspecified”.

Install a *package*

Abbreviation:	AI1
Priority:	Essential
Description:	The *node administrator* needs to install *packages*. The *packages* may come in form of files or may be available on *distribution servers*.
Status:	Done. A *distribution server* is not a dedicated server. Packages are available directly from a *repository*.

Install an *update package*

Abbreviation:	AI2
Priority:	Essential
Description:	The *node administrator* needs to update *packages*. The *update packages* make come in form of files or may be available on *distribution servers*.
Status:	Done, but the tools do not distinguish between regular packages and update packages yet.

Download required third party bundles

Abbreviation:	AI3
Priority:	High value
Description:	During the installation of a new *package* or an *update package*, the *node administrator* should be given an option to let the system download required third party *entities* or *packages* from *distributions servers* and *repositories* over the Internet.
Status:	No support. Dependencies between packages can not arise with the current package implementation.

Install specific updates

Abbreviation:	AI4
Priority:	Follow on
Description:	The *node administrator* should be allowed to install a specific version of a *package* (update), even if a newer version of the *package* is available (and known to the system).
Status:	Done.

Automatic update download and installation

Abbreviation:	AI5
Priority:	Follow on
Description:	Provide a tool that automatically finds *package updates* on known *distribution servers*, downloads them, stores them in the local package repository and installs them. All these functions should be provided as fully automatic. The interface of this tool should allow scheduling in Cron or Windows Scheduler.
Status:	No support. The tools even do not distinguish between regular packages and update packages yet.

Partial installation of a *package*

Abbreviation:	AI6
Priority:	Exempt
Description:	Provide functions to install only some *entities* from a *package.*
Reason:	This function would damage the applications in the *repository* or even the *repository* itself.

Manage multiple versions of a similarly named *package*

Abbreviation:	AI7
Priority:	Exempt
Description:	The system should allow to install and manage multiple versions of a *package* with the same name. That means multiple versions of the same *package* and also different *packages* with the same name (eg. from different distributors).
Reason:	If the package name is not an unique identifier of a *package*, what should be the identifier then?

Preserve customizations to files across updates

Abbreviation:	AI8
Priority:	Exempt
Description:	The changes that the *node administrator* made to a file should be preserved when an update package (that modifies the same file) is installed.
Reason:	It's not clear how files fit in *SOFA 2* ideas. The meta-model does not handle them.

Uninstall a *package*

Abbreviation:	AR1
Priority:	Follow on
Description:	The *node administrator* should be provided with functions for manually selecting a *package* to remove and be notified of dependencies prior to removal. If there are some *packages*, that depend on this *package*, he or she should have an option to cancel or remove it with all the dependent *packages.* The *package* should be removed with all updates. Conflicting situation is, when an *entity* from the *package* that is to be removed is installed also with another *package.*
Status:	Partially done. Each of the *(update) packages* have to be uninstalled separately.

Uninstall an *update package*

Abbreviation:	AR2
Priority:	Follow on
Description:	As in requirement *AR1*, the *node administrator* should be provided with functions for removing an *update package*, leaving the updated *package* in the state it was just before applying the *update package*.
Status:	Done. *Update package* can be removed, because the tools treat it like a normal *package*.

Remove *package* from local *package repository*

Abbreviation:	AR3
Priority:	Exempt
Description:	It should be possible to remove the contents of a *package* from the local *package repository*. For the purpose of searching and manipulating functions, the meta-data of the *package* should remain in the *package repository* until all *entities* from the *package* are completely removed from the *SOFAnode*.
Reason:	Local *package repository* was dropped because of simplicity and consistency reasons.

Uninstall a *package* without source

Abbreviation:	AR4
Priority:	Exempt
Description:	As in requirement *AR1*, the *node administrator* should be allowed to remove a *package* that is not in the local *package repository* and its source (*distribution server* or file) is not available. For this function, the meta-data of the *package* have to be stored in the local *package repository*.
Reason:	Local *package repository* was dropped because of simplicity and consistency reasons.

Uninstall an *update package* without source

Abbreviation:	AR5
Priority:	Exempt
Description:	As in requirement *AR2*, the *node administrator* should be allowed to remove an *update package* that is not in the local *package repository* and its source (*distribution server* or file) is not available. For this function, the meta-data of the *update package* have to be stored in the local *package repository*.
Reason:	Local *package repository* was dropped because of simplicity and consistency reasons.

Remove not used *entities* after update

Abbreviation:	AR6
Priority:	Follow on
Description:	After a *package* is updated or upon request, the *node administrator* should be asked whether he or she wants the system to remove *entities* that are no longer used (in scope of the *entities* brought by the base *package* and the update chain preceding the last update). He or she should also be warned if there are some customizations in such files. This is in conflict with requirement *AR5*. Some updates then may become unremovable, if some of the *packages* from the update chain or the base *package* are not available.
Status:	No support. The tools do not distinguish between regular packages and update packages yet. File will not be supported at all.

Remove not used files after update

Abbreviation:	AR7
Priority:	Exempt
Description:	After a *package* is updated or upon request,the *node administrator* should be asked whether he or she wants the system to remove files that are no longer used (in scope of the files brought by the base *package* and the update chain preceding the last update). This is in conflict with requirement *AR5*. Some updates then may become unremovable, if some of the *packages* from the update chain or the base *package* are not available.
Reason:	It's not clear how files fit in *SOFA 2* ideas. The meta-model does not handle them.

A.4.3 Package transfer and distribution

Create an off-line distributable file

Abbreviation:	D1
Priority:	Essential
Description:	The *packages* should be created in form of a file if requested. The file format should allow all usual distribution means of files, for example on CD, posting on a web etc. The distribution of the file is then left on the *distribution manager* and is not covered by the system.
Status:	Done.

Deliver *packages* in a compressed format

Abbreviation:	D2
Priority:	High value
Description:	The *packages* should be distributed in compressed form (e.g. zip) to save disk space and/or network bandwidth.
Status:	Done.

Direct connection to both sides

Abbreviation:	D3
Priority:	High value
Description:	The system should allow the *distribution manager* to connect via direct connection to both source (where the *package* is to be created) and target (where the *package* should be installed) *SOFAnodes*, if both of them are accessible through the Internet. The *distribution manager* creates a *package* the same way as when it should be distributed by file, except the *package* is not stored on file system. The *package* is then transferred on the target *SOFAnode* and installed in the same manner as if the *package* file was supplied. The *distribution manager* therefore fulfils also the roles of the *assembler* and *node administrator*. He or she is presented the same information by the system, as when creating and installing a *package* that is distributed by file.
Status:	Done.

Direct connection with one supplier and multiple receivers

Abbreviation:	D4
Priority:	Follow on
Description:	The system should allow the *distribution manager* to connect via direct connection to one source *SOFAnode* (where the *package* is to be created) and multiple target *SOFAnodes* (where the *package* should be installed), if all of them are accessible through the Internet. The creation and installation of the *package* should follow the same rulee as described in requirement *D3*. The installations on target machines should be sequential in order not to flood the *distribution manager* by information from different sources at once.
Status:	Not supported.

Post *packages* and let the *node administrators* pull them

Abbreviation:	D5
Priority:	Follow on
Description:	After the *package* is created, the system should optionally store the *package* in such a manner, that it is accessible by *node administrators* over the Internet. This might by realized for example as a running server instance that is waiting for the connections. The *distribution manager* is provided with a text description how to connect to the new server. He or she distributes this information to *node administrators*, which pull the *package(s)* from the server. The place, where the *packages* are posted for internet access will be called *package repository* in the following paragraphs.
Status:	Not supported.

Authorized access

Abbreviation:	D5.1
Priority:	Follow on . Essential
Description:	The *node administrators* have to authorize themselves by username and password for access to the *package repository.*
Status:	The *package repository* is not implemented.

Limit access to contents by *node administrator*

Abbreviation:	D5.2
Priority:	Follow on . High value
Description:	The *node administrators* should have access to only limited content of the *package repository.* The limits should be set by the *distribution manager* for *node administrators* according to the username of their company. The smallest unit of access to be limited should be a *package.*
Status:	The *package repository* is not implemented.

Show *package* meta-data to *node administrator*

Abbreviation:	D5.3
Priority:	Follow on . High value
Description:	The *node administrators* should have an option to list the meta-data of a *package* prior to pulling it from the *package repository.*
Status:	The *package repository* is not implemented.

Show *package* content and meta-data to *distribution manager*

Abbreviation:	D5.4
Priority:	Follow on . High value
Description:	The *distribution manager* should have an option to list the content and meta-data of selected *package* without pulling the whole *package* from the *package repository.*
Status:	The *package repository* is not implemented.

Public access

Abbreviation:	D5.5
Priority:	Follow on . Follow on
Description:	The *package repository* should allow access without authorization (username and password). Such access should be restricted to some *packages* defined by the *distribution manager.*
Status:	The *package repository* is not implemented.

Report number of downloads by a *node administrator*

Abbreviation:	D5.6
Priority:	Follow on . Follow on
Description:	The *distribution manager* should have an option to find out, how many times was a *package* downloaded and by whom. The system also should aggregate the information by *node administrators* or companies.
Status:	The *package repository* is not implemented.

Report results of the installation to *distribution manager*

Abbreviation:	D5.7
Priority:	Follow on . Count with
Description:	The report containing number of downloads from requirement *D5.6* should also provide the number or percentage of successful and failed installations.
Status:	The *package repository* is not implemented.

Send email notifications to *node administrators*/companies

Abbreviation:	D5.8
Priority:	Follow on . Count with
Description:	The system should send a notification, when a new *package* is available for a *node administrator* or a company. It's upon *distribution manager*'s decision, whether the system will send the notification. This might be possibly overridden by the preferences of each *node administrator*.
Status:	The *package repository* is not implemented.

Licensing and license enforcement

Abbreviation:	D6
Priority:	Exempt
Description:	The system should support various means of licensing (eg. free copying, as a book, limiting number or running applications, ...). Users or *node administrators* should be forced to obey the license agreement by the system. The license enforcement shouldn't be easy to break.
Reason:	Licensing is out of scope, it's not the aim of the system. Real license enforcement is not practically possible in an open source system.

A.4.4 Non-functional

Complete command line functionality

Abbreviation:	NF1
Priority:	Essential
Description:	Virtually all of the basic *package* management and distribution operations must be available through CLI mode.
Status:	Done.

GUI

Abbreviation:	NF2
Priority:	Follow on
Description:	The basic *package* management and distribution operations must be available through GUI.
Status:	Done.

GUI works with *SOFA 2 IDE*

Abbreviation:	NF2.1
Priority:	Follow on . Essential
Description:	The GUI must be an Eclipse plug-in that operates side by side with *SOFA 2 IDE* or as an optional add-on to it.
Status:	Done.

GUI provides *package* creation and distribution through GUI

Abbreviation:	NF2.2
Priority:	Follow on . Essential
Description:	The basic *package* creation and distribution operations must be available through GUI.
Status:	Done.

Package installation, monitoring and uninstallation

Abbreviation:	NF2.3
Priority:	Follow on . Follow on
Description:	The GUI must provide the functionality to install a *package*. It should also provide an interface to search the installed *packages* by various meta-data and uninstall a *package* upon request.
Status:	Done.

GUI works with *SOFA 2 MConsole*

Abbreviation:	NF2.4
Priority:	Follow on . High value
Description:	The GUI must be an Eclipse plug-in that operates side by side with *SOFA 2 IDE* or as an optional add-on to it.
Status:	Done.

Minimize UIs required for ***package*** management functions

Abbreviation:	NF3
Priority:	High value
Description:	Preferably, a single GUI and a single CLI should be all that is needed to carry out basic *package* management and update operations.
Status:	Done. The SOFA IDE and Cushion are the only UIs used.

High quality error messages

Abbreviation:	NF4
Priority:	High value
Description:	The program should report what went wrong. If the error is expected during development, the system should also report what is the expected cause. The location and possibly full stack trace should be written only to a log file, not on a console.
Status:	Done.

Relevant information should be logged to a log file

Abbreviation:	NF5
Priority:	High value
Description:	The *SOFA 2* logging system (*log4j*) should be used. The logging configuration/settings should be shared with *SOFA 2*.
Status:	Done.

SOFA 2 autoconfiguration as default.

Abbreviation:	NF6
Priority:	High value
Description:	The tools should be aware of *SOFA 2 autoconfiguration* and use it as a default for connecting to *repository* whenever possible.
Status:	Not supported. The SOFA IDE and Cushion are not aware of SOFA 2 autoconfiguration.

Sufficient busy/progress feedback

Abbreviation:	NF7
Priority:	Follow on
Description:	Both the CLI and GUI should report some progress for all the actions that are expected to take 3 seconds or longer. The progress bar should be used where applicable, actions done should be reported elsewhere.
Status:	Done.

B. Contents of the Enclosed DVD

This thesis is accompanied by a DVD-ROM containing binaries and source code of the prototype implementation. The DVD-ROM is organized as follows:

readme.txt A description of the contents of the enclosed DVD-ROM and instructions for using it.

master-thesis.pdf An electronic version of this thesis in PDF format.

bin/ Binary distribution of the SOFAnet 2 prototype implementation.

- **bin/sofa/** Prepared SOFA 2 environment including a repository filled with some examples. Cushion is included too.
- **bin/sofa-alternate-repo/** An empty SOFA 2 repository configured to run on an alternative port. A distribution of Cushion configured to use the alternative port is included too. This repository should be able to run on the same machine with the SOFA 2 environment mentioned above. Included for testing purposes.
- **bin/eclipse/** Binary form of Eclipse plug-ins that build up the SOFA 2 IDE with SOFAnet 2. Please read `readme.txt` for details on installation.

src/ Source code of the prototype implementation.