Automated Reuse of System Functionality Information for FPA and System Specification

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE ENGINEERING

by

Iljaas Habiboellah
born in Purmerend, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Capgemini Business Application
Services B.V.
Staalmeesterslaan 410
Amsterdam, the Netherlands
www.capgemini.nl

Automated Reuse of System Functionality Information for FPA and System Specification

Author: Iljaas Habiboellah
Student id: 1053035
Email: iljaas.habiboellah@gmail.com

Abstract

In the process of creating an information system for a customer, the function point analysis workflow and the system specification workflow fail to take advantage of each others knowledge gained during these workflows. Analyzing the functionality of a system is usually a time consuming task, which is currently performed for each workflow separately. In this research we created SFRL (System Functionality Recording Language), a means for formally recording the knowledge gained during such an analysis. By formally recording this knowledge, it becomes easier to exchange information between workflows. To make the information useful we created mappings from SFRL to function point counts based on the NESMA specification and to an initial system specification based on the UML components method. Also, we mapped the information in a UML components specification back to SFRL. These mappings were implemented in a model transformation language. This way we enabled ourselves to perform an automated function point count based on the UML components specification of a system. Furthermore, we introduced a means for performing an automated functional completeness evaluation using SFRL.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. P.G. Kluit, Faculty EEMCS, TU Delft
Company supervisors:	Ing. J.D. Gerbrandy, MBA, Capgemini BAS
	Dhr. T. Thijssen, Capgemini BAS
Committee Member:	Ir. B.R. Sodoyer, Faculty EEMCS, TU Delft

Acknowledgements

This thesis is the result of my MSc project at the Software Engineering Research Group (SERG) at the Delft University of Technology. The project was carried out at the research and development department of Capgemini BAS.
During this project I was supervised by Jelle Gerbrandy, Theo Thijssen and Peter Kluit. I would now like to take the opportunity to thank them for their guidance, encouragement and for their patience during this project.
I want to thank Jelle for his sharp and detailed feedback and suggestions and for always taking the time to answer my questions. I really admire the fact that he has such a wide range of knowledge.
I'm grateful to Theo for sharing his expertise in the area of function point analysis, for spreading his enthusiasm and for keeping me motivated.
I want to thank Peter for his constructive feedback, for sharing his insights, for always encouraging me to achieve progress and for being a great teacher.

A special thanks goes to my loving parents for their support, their patience and for providing me with the opportunity of getting this level of education. They have always been there for me and I could not have wished for any better parents.

Iljaas Habiboellah
Delft, the Netherlands
May 29, 2009

Contents

List of Figures

Chapter 1

Introduction

An information system development project consists of various phases. Companies exist which create information systems for customers. From the customer's point of view, at the beginning of the project requirements have to be defined. The next step is to get estimates from one or more companies of the costs of such a project, based on the requirements as specified by the customer.
Let's say a company named Capgemini Business Application Services (Capgemini BAS) exists which creates information systems and this company is contacted by a customer. We will now switch to the company's point of view. The customer provides documentation in which their requirements are recorded. Based on these requirements the size of the information system to be built has to be estimated as a part of determining the costs of the project. Based on this estimate an offer can be made to the customer.

1.1 Function Point Analysis

Function Point Analysis (FPA) was developed as a result of productivity research in a large number of projects. FPA introduces a unit, the function point, which can be used to estimate the functional size of an automated information system to be developed or to be maintained [11]. Several methods to perform a function point analysis exist. The NESMA (NEderlandse Software Metrieken Gebruikers Associatie) is an institution which provides a method for measuring the functional size of a system. Within Capgemini BAS the method as provided by the NESMA is used for performing an FPA. In Chapter 2 we will describe this method in more detail.

1.2 System specification

When the size of the system is estimated, the costs for realizing a system of such a size can be estimated and an offer can be made to the customer. If the customer agrees with the offer, the project is assigned to the company, which will then further develop the information

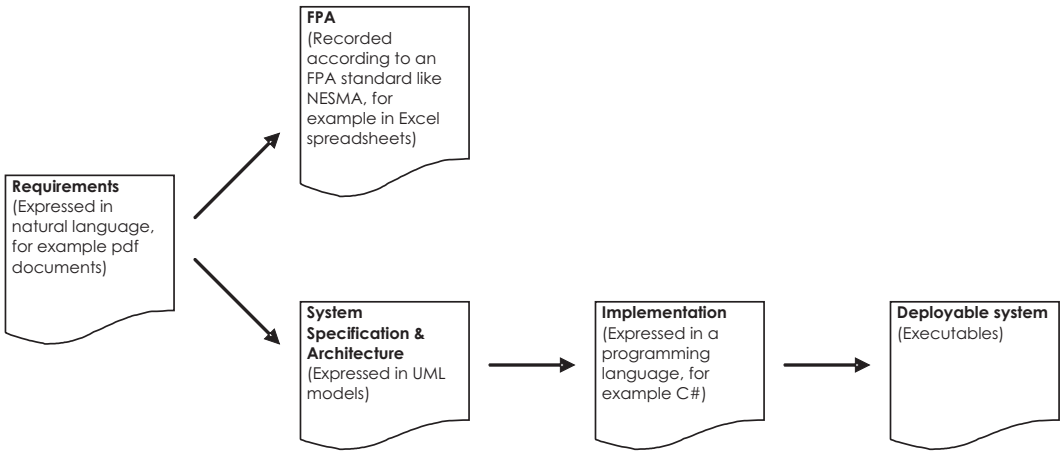

Figure 1.1: Simplified depiction of artifacts in a software development project

system. In this process the system is specified and an architecture is designed, the system is implemented in some programming language and as a result the product evolves into a deployable system (see Figure 1.1).
Several methods exist that can be used in order to create a system specification. The method used in this research is the UML Components method as described by Cheesman and Daniels [4]. In Chapter 2 we will give a brief description of this method.

1.3 Problem description

The requirements for the system to be built are specified in natural language. When the size of the project has to be estimated, one or more function point analysts perform an analysis of the system based on the functionality described in the requirements documentation.
When the system has to be built, the system developers also perform an analysis of the system based on the same requirements documentation. With the gained knowledge a system specification is built.

The first problem that can be noticed here is the fact that the result of the work carried out by function point analysts is hardly used by the system developers. Instead, an analysis of the same documentation takes place all over again. The function point analysis, however, contains information about the system that could very well be used as a stepping stone for the system specification. One of the aims of this thesis is to provide a means to take advantage of the knowledge gained during the first analysis when creating a system specification.

Due to progress and feedback during the development process of the system, the understanding of the requirements improves and changes may occur in the customer's desires. To

get a grip on the project's costs in proportion to the achieved results, the function points will be recounted at least once by function point analysts using updated documentation.
Another problem that comes to light here is the fact that currently the information that is recorded during the system specification remains unused in this recounting process. As one might understand, while the functionality of the system is presented orderly in the system specification, recounting the function points from documentation in natural language is a suboptimal and very time consuming way of keeping track of the system size. Another aim of this research is to take advantage of the knowledge recorded in the system specification when recounting the function points in the system.

To summarize the problem description: In the process of creating an information system for a customer, the function point analysis workflow and the system specification workflow fail to take advantage of each others knowledge gained during these workflows.

1.4 Proposed solution

The result of a function point analysis is unmistakably related to the system specification in some way, because the function points counted during the analysis represent the estimated size of the functionality devised in the system specification. During the first analysis by function point analysts, the functionality has to be identified in order to count the size. During the second analysis by system developers, the functionality has to be identified in order to create a system specification.

The proposed solution to the aforementioned problems comes down to storing the information gathered during the first analysis of the system to be built in an intermediate format. Using the stored information, the size of the system can be determined and a base for the system specification can be created. In order to do so, mappings between the intermediate format and the function point analysis and mappings between the intermediate format and the system specification have to be defined. By means of a model transformation language,

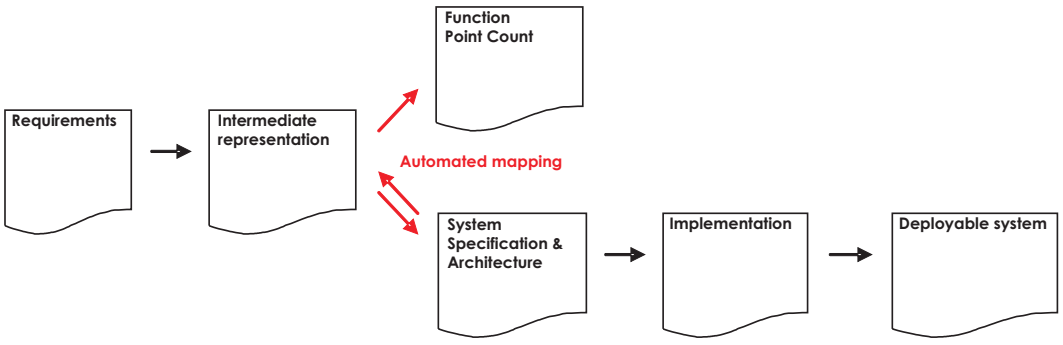

Figure 1.2: Proposed solution

these mappings can be captured in rules which can be matched and applied automatically to a source model (see Figure 1.2). The intermediate format is a means for recording the information in such a way that is interpretable for the model transformation engine. The intermediate format will be given the name SFRL (System Functionality Recording Language).

This solution solves the problems mentioned above. Firstly, the analysis of the requirements in natural language does not take place for the function point count and the system specification separately. Secondly, this approach provides an automated path from the system specification toward a function point count. Therefore, for a recount of the function points in a system there is no need to analyze updated documentation in natural language. Instead, a function point recount can be generated automatically.

Another advantage of this solution is the fact that recounts performed by function point analysts can not always be compared properly, because with each recount deviations may occur in the way function point analysts evaluate the requirements. Generated function point recounts, however, can be compared properly, because these deviations do not occur. Each time, over and over again, the function point counts are performed according to the exact same set of rules. This does not mean that the way of establishing the system size is more perfect, but if there is an impurity in a function point count, it will be present in every function point count in the exact same way, which makes it easier to compare the results of generated counts to each other.

Furthermore, formally recording the information about the functionality of a system can be useful for other purposes such as automatically evaluating the completeness of the specified functionality (see Chapter 3.6).

1.5 Research questions

In this thesis several questions are answered. Our main research goal is to create a process in which knowledge gained during the function point analysis can be shared so it can be useful for creating a system specification and vice versa.

In this thesis the following questions are answered:

1. What should the intermediate representation for recording the system functionality (SFRL) look like? What entities need to be recorded?
    a) Which details about a system have to be known in order to determine the size of the system?
    b) What artifacts are recorded in a UML components specification?
    c) Can we deal with different levels of details about the system during the system development process?

2. Are the similarities between the NESMA standard and the UML components method sufficiently captured in order to map both onto each other?

3. Are the created mappings sustainable?

4. Can this solution be put to use in the software engineering process? What are the consequences for the way in which function point analysts and system specifiers have to work?

5. Are there additional benefits to this approach?

1.6 Research approach

At the end of this project we need to have an answer to the questions above. In this section we describe our approach for how to get the answers to these questions.

Part of this research is the creation of a new language called SFRL. To be able to create this language we need to know what we want to record in this language. This is what the first question is about. In order to know what the language should look like we need to know what we want to express in this language. In order to know what we want to express in this language we need to get a clear view of the purpose of the language. In this section we already described how we want to use the language: as an intermediate format for exchanging information between a function point analysis and a system specification.
We have to get acquainted with the standard used for function point counting (in our case the NESMA standard) and the method used for specifying the system (in our case the UML components method). Once we know what information is involved in these methods, we can create an overview of the information that needs to be recorded in SFRL.

Even though we do not directly map information from a NESMA function point analysis to a UML components model, we do investigate upon the links between the NESMA standard and the UML components method, because only then we come across the obstacles in the path from a function point count to a UML components specification. This way we know what we should take into account when creating SFRL. Moreover, SFRL is based on information from the NESMA standard and this way we can find out what we need to adapt in order to make the mapping to UML components possible.

The next step is to come up with a metamodel according to which SFRL models can be created and a metamodel for expressing function point counts. We can then create model transformations which implement mappings from the SFRL metamodel to the function point count metamodel, transformations which map SFRL models to UML components models and vice versa.
In order to create SFRL models a convenient user interface is desirable, since we do not want to write XML code for the SFRL model manually. Therefore we come up with such a user interface.

Furthermore, based on the information in an SFRL model a functional completeness evaluation can be performed. This functional completeness evaluation can be automated by creating a model transformation for this purpose. The target model of this transformation is a model in which the result of the evaluation is recorded. A metamodel for this target model has to be defined.
Finally, we evaluate the usability and usefulness of SFRL by means of some test systems.

1.7 Outline

In Chapter 2 we describe the NESMA standard used for function point counting and the UML components method used for specifying a system in UML. Furthermore, we describe how these are related to each other.
In Chapter 3 we devise a case for exploring the relationships between function point analysis and system specification and the entities that are desirable for recording in the intermediate representation. We come up with a metamodel for SFRL and for expressing function point counts. Moreover, we provide ourselves with a simple user interface for creating an SFRL model. In this chapter we also describe the mappings from SFRL to function point counts and the mappings from SFRL to UML components and vice versa. Furthermore we come up with a method to evaluate the functional completeness of the system based on the information in the SFRL model and a metamodel for expressing the functional completeness evaluation.
In Chapter 4 we evaluate the usefulness and usability of SFRL and the created model transformations using some example systems. First we evaluate the way of creating SFRL models. Then we take the initial SFRL models of the test systems, transform these into UML components models which are then transformed back to SFRL models and compare the initial SFRL models to the regenerated SFRL models. Furthermore, we transform a manually created UML components model to SFRL and then transform it back to a UML components model and then compare the initial UML components model to the regenerated UML components model. Also, we evaluate the method for determining the functional completeness of a system.
We conclude this report in Chapter 5.

Chapter 2

Methods involved

In this chapter we will discuss the FPA (Function Point Analysis) method and the system specification method involved in this thesis and discuss the connection between these.

2.1 Function Point Analysis: NESMA

The method used at Capgemini BAS to count function points in information systems is the method as specified by the NESMA (ISO/IEC 24570:2005). With this method the size of the functionality in a system is measured, based on what the system does and not on how the system realizes the functionality. In this section we will describe the information in the NESMA standard in more detail.

Counting function points according to the NESMA standard consists of four main parts:

- identify functions and determine function types
- assign complexity to functions
- assign function points to functions based on complexity
- add up function points of individiual functions

2.1.1 Function types

Five types of functions are defined in function point analysis. The function types are categorized in two main groups, namely the data function types and the transactional function types. The data function types can either concern internal data or external data. The transactional function types can either be inputs, outputs or inquiries (see Figure 2.1).

The internal logical files (ILF) concerns data which is used by the application to be counted and is also maintained by that application. External interface files (EIF) concerns

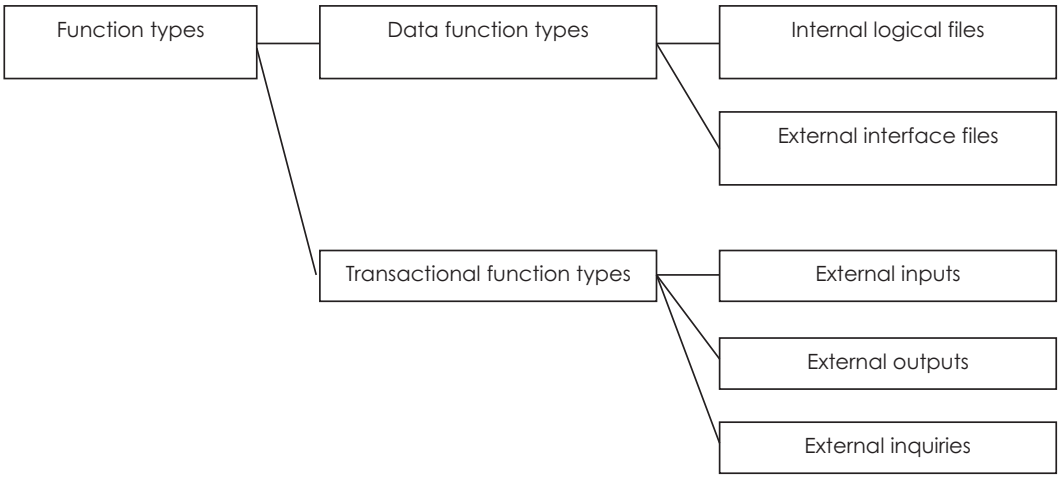

Figure 2.1: Function types

data which is directly available to and used by the application to be counted, but maintained by another application. Additionally, entity types may be identified as FPA tables if:

- the entity type contains one and only one item of data
- the entity type contains only data that is constant
- the entity type consists of a key and one or more explanatory descriptions, provided that the explanations are of a similar kind.
- the entity types contains boundary values, algorithms, minimum and maximum values, provided that the key is single.

FPA tables that are maintained by the application to be counted are called FPA tables ILF. FPA tables that are maintained by another application are called FPA tables EIF.

Transactional function types represent functionality that must be seen as unique, elementary processes which leave the system in a consistent state after the function has been invoked. External input (EI) is functionality in which data and/or control information is entered into the application from outside the application. An external output (EO) is functionality in which output is recognized by the user that crosses the system boundary. The size of this output may vary and/or further data processing may be required to establish the output. An external inquiry (EQ) is a combination of input and output in which the size of the output is determined and no further data processing is done to establish the output as a result of the input.

2.1.2 Complexity of functions

The functions in an information system may vary in complexity. The NESMA standard provides three levels of complexity for each function type: low, average or high. Depending

on their complexity, function points are assigned to each function (see Figure 2.2). The way of determining the complexity of a function depends on the type of function point count used (see next section).

	ILF	EIF	EI	EO	EQ
Low	7	5	3	4	3
Average	10	7	4	5	4
High	15	10	6	7	6

ILF = Internal Logical File
EIF = External Interface File
EI = External Input
EO = External Output
EQ = External Inquiry

Figure 2.2: Function points

2.1.3 Types of function point counts

Three different types of function point counts are defined in the NESMA standard.

- indicative
- estimated
- detailed

Depending on the level of detail of the documentation about the information system, a type of function point count can be chosen. The indicative function point counting type aims to provide a rough indication of the system's size, based on the data function types identified in a system. For each data function type a default number of function points is assigned, based on a default complexity of the data function type and a default number of transactional function types with default complexity. Based on a conceptual data model, each ILF is 35 function points and each EIF is 15 function points. FPA tables ILF are counted all together as one ILF and FPA tables EIF are counted all together as one EIF.

An estimated function point count uses more information for the function point count and is, therefore, more accurate. The complexity for each function type is still a default value, but the transactional function types are actually identified instead of taking a default number of transactional function types per data function type into account. The data function types have a low complexity by default. The transactional function types have an avarage complexity by default. Using Figure 2.2 function points can be assigned to each function type.

A detailed function point count is the most accurate of the function point counts. With this type of count the function types have no default complexity. Instead, the complexity

of data function types is determined by the number of data element types (DETs) and the number of record types (RETs) in the data function type.

The complexity of an ILF is determined as follows:

RET \ DET	1-19	20-50	51+
1	Low	Low	Average
2-5	Low	Average	High
6+	Average	High	High

Figure 2.3: Complexity of an ILF

The complexity of an EIF is determined as follows:

RET \ DET	1-19	20-50	51+
1	Low	Low	Average
2-5	Low	Average	High
6+	Average	High	High

Figure 2.4: Complexity of an EIF

The complexity of transactional function point types is determined by the number of data element types and the number of file types referenced (FTR).

The complexity of an EI is determined as follows:

FTR \ DET	1-4	5-15	16+
0-1	Low	Low	Average
2	Low	Average	High
3+	Average	High	High

Figure 2.5: Complexity of an EI

The complexity of an EO is determined as follows:

FTR \ DET	1-5	6-19	20+
0-1	Low	Low	Average
2-3	Low	Average	High
4+	Average	High	High

Figure 2.6: Complexity of an EO

The complexity of an EQ is determined as follows:
- The complexity of the input part is determined according to Figure 2.5.
- The complexity of the output part is determined according to Figure 2.6.
The complexity of the EQ will be the greater of these two.

2.1.4 Required information for function point counts

The information required for function point counts depends on the type of function point count used.

Indicative function point count

For the indicative function point count we need to identify the data function types. In order to do so, we need a conceptual data model or a normalized data model. To make a distinction between ILFs and EIFs, we need an indication of whether the data function types are maintained by the system to be counted or by another system, because the first results in a greater amount of function points than the latter.

Estimated function point count

For the estimated function point count more information is required. On top of the information required for an indicative function point count, we also need a model which shows the system functions with their incoming and outgoing information flows from and to the environment of the system.

Detailed function point count

For a detailed function point count, again, extra information is required on top of the information required for an estimated function point count. We need information, based upon which we can determine the complexity of the function types. This means that the record types and the data element types of logical files must be known. Furthermore, the incoming and outgoing data flows of the application needs to be detailed up to the level of data ele-

ment types and the logical files addressed by the transactional functions need to be known in order to determine the complexity of the functions.

2.2 System specification: UML Components Method

The UML Components method as described by Cheesman and Daniels [4] is an approach used at Capgemini BAS for specifying software components for information system architectures. In this section we will give a brief overview of the UML Components Method and the artifacts involved.

The specification of the functionality of a component may be split across one or more so called interfaces. This means that dependencies can be restricted to individual interfaces rather than encompass the whole component specification. Components in a system can be substituted by any other components with a different component specification, provided that the interfaces used by the clients remain the same.

According to the UML Components method the architecture of an application can be split up in 4 different layers (see Figure 2.7). The bottom two layers form the server part, while the upper two layers form the client part. The server part forms the system, which is independent of the way user interaction is implemented. If user dialog logic and a user interface are added to the system, an application is formed. The UML Components method focusses on the system layers.

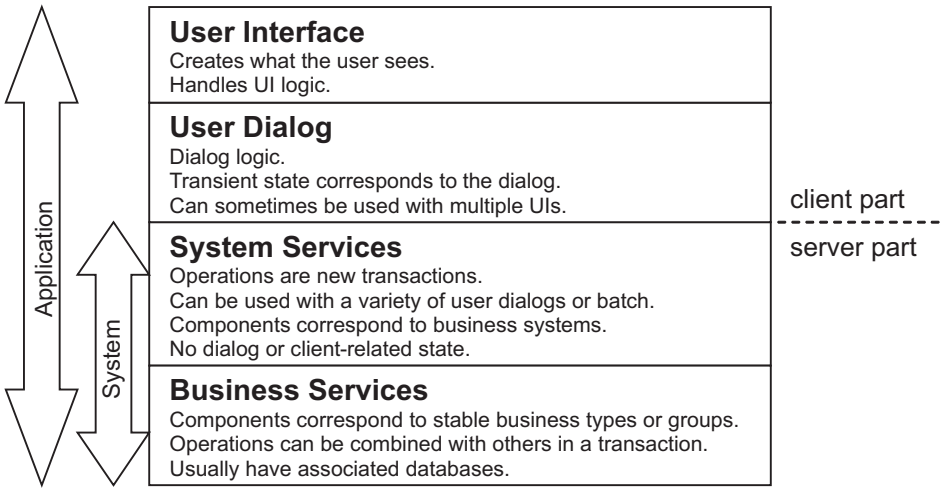

Figure 2.7: Layers in an application's architecture [4]

2.2.1 Workflows and artifacts

The UML Components method adopts the definition of a workflow as defined by Kruchten in his book on RUP [8], ”a sequence of activities that produces a result of observable value”. The deliverables that carry information between workflows are called artifacts. The UML components Method focuses heavily on the requirements and the specification workflows and the artifacts involved in these.

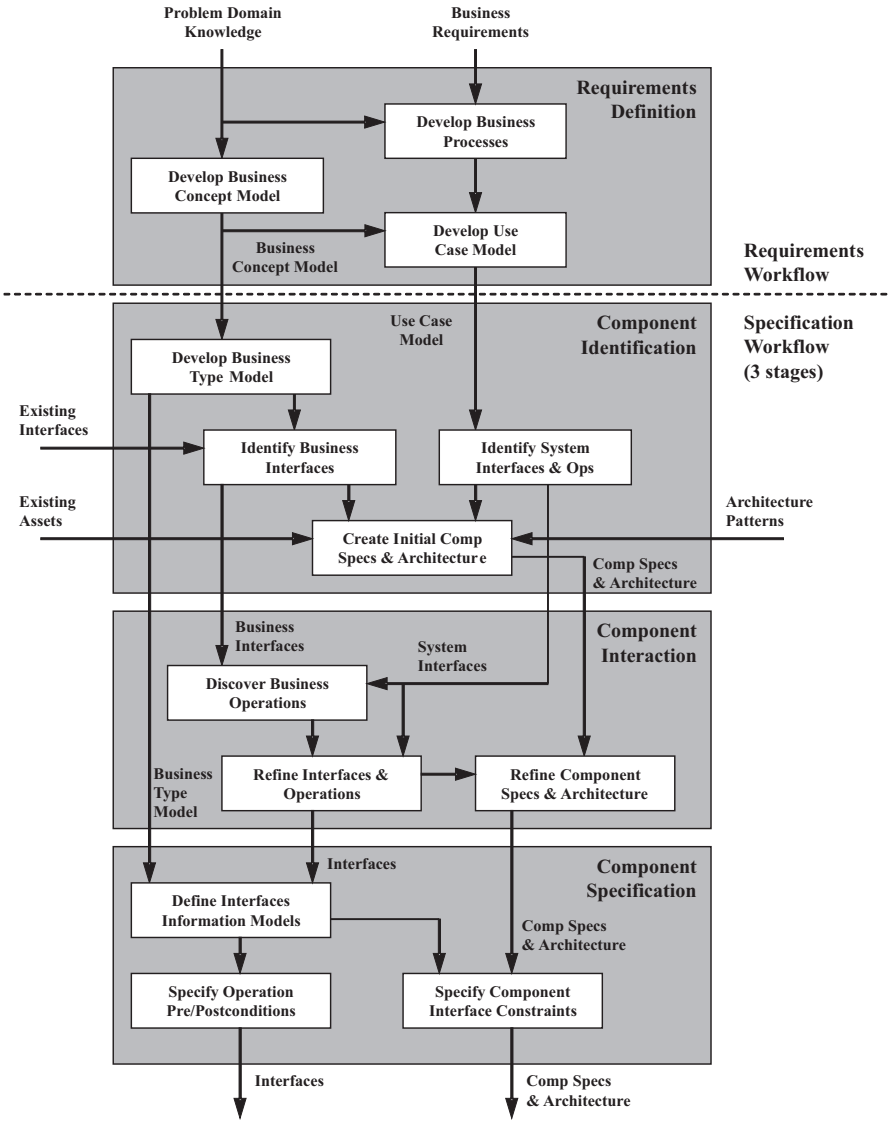

Figure 2.8: Requirements and Specification workflows

As one can see in Figure 2.8, during the requirements workflow a business concept model and a use case model are created. These will be used as input for the specification workflow, which consists of three stages: component identification, component interaction and component specification.
In the component identification stage, the business concept model is scoped and refined into a business type model. In this model only the concepts from the business concept model which are the system's responsibility remain. Furthermore, Business interfaces are identified. From the use case model, initial system interfaces and operations are identified.
As the name suggests, during the component interaction stage the interaction between components is elaborated. The components interact by means of operations provided by the interfaces. In this stage the initial system interfaces and operations are refined and business operations are discovered. Furthermore, decisions regarding referential integrity have to be made during this stage. It has to be decided whether the integrity of references has to be guaranteed and, if so, how that can be achieved.
In the final stage of the specification workflow all the final details of the usage contracts and the realization contracts of the components are elaborated. The usage contract is a contract between the component object and its clients. The realization contract concerns additional specification information for the component implementer and assembler.

The artifacts created during these workflows are stored in UML diagrams. These artifacts are organized according to the package hierarchy as seen in Figure 2.9.

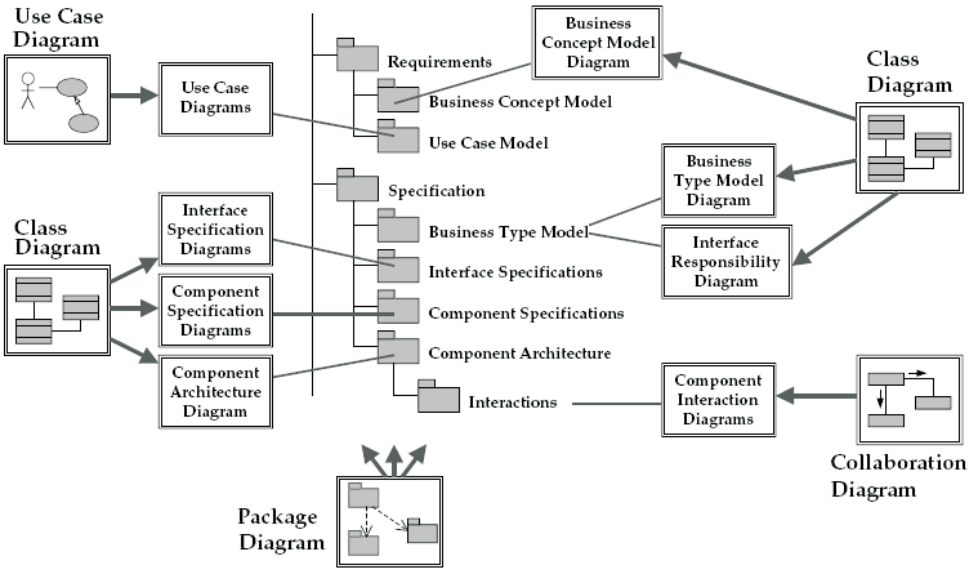

Figure 2.9: Component specification diagrams [4]

2.3 Connection between FPA and System specification

In the proposed solution it is suggested that there is a connection between the information in a function point analysis and a system specification. Since both are based on the same documentation about the same system, this seems like a credible premise. In this section we point the similarities we came across when describing both methods.

Both the function point analysis and the system specification describe the system's functionality. The NESMA standard focuses on what functionality is present in an information system and not on how the functionality is implemented. The UML components standard specifies the functionality of system components by means of use cases and eventually interfaces. Here also, it is not important to specify how the functionality is offered by the component to a client, but only what functionality is offered through the interfaces. After all, it was stated that as long as the interfaces remain the same, the components (i.e. the way of implementing the functionality provided through the interface) are replaceable.

In Section 2.1.4 we stated that even for the most simple type of function point count a model from which the logical files can be determined should be available, for example a conceptual data model. Such a model can be found in the business concept model in the UML components specification.
The NESMA standard makes a distinction between data function types that are maintained by the system in question and data function types that are maintained by an external system. As stated in Section 2.2.1, in the UML Components method, the business concept model will be scoped and only concepts for which the system in question is responsible will be maintained in the business type model. By comparing these two models it becomes clear which information used by the system is maintained internally and which information is maintained by external systems. So the UML Components method also supports this distinction, albeit in a less obvious way.

As we can see here, there is definitely a connection between a function point analysis and a system specification according to the UML Components method. Without doubt, there are also be differences between the methods, like dissimilarities in detail levels between entities, that we need to overcome.
A good example of this is when comparing the way transactional functionality is organized in both methods. In the function point analysis we are dealing with transactional function types, which are not grouped in any way, while in the UML components method we are dealing with use cases and use case descriptions, the latter of which consist of stimuli and responses which are grouped together in the use cases. These use cases and descriptions will in a later stage be converted into system interfaces in which operations are grouped together.
Apart from the problem of functionality being grouped on the one side while not being grouped on the other side, we already had to deal with the ”perennial problem with use cases” of deciding their scope and size. According to Cheesman and Daniels [4], there seems to be no consensus on this issue, but they give their view on the problem: ”To a first

approximation we can say that a use case is smaller than a business process but larger than a single operation on a single component. The purpose of a use case is to meet the immediate goal of an actor, such as placing an order or checking a bank account balance. It includes everything that can be done now or nearly now by the system to meet the goal". This means that functionality required for multiple purposes may return in multiple use cases on the UML components side of the story, while that functionality will only be modeled once as a transactional function type on the FPA side.
According to Cockburn [5] three goal levels can be distinguished, namely the summary goals level, user goals level and the subfunctions level. The summary goals provide a context in which user goals operate. These use cases visualize the goals of a system for the organization. This level is too high [12] to determine transactional function types, which represent elementary functionality [11]. The user goals visualize the goals of a system for the user of that system. Subfunction level goals can not be positioned as goals of actors, but are required to achieve user goals. For example, to check whether a room in a hotel is free on a given date is not a goal of an actor, but is part of, for example, making a reservation. Making a reservation is a user goal, which is part of the summary goal, managing reservations.
Luckily the UML components method provides a way for modeling functionality that is used multiple times only once. Namely by means of including and extending use cases: "...in UML terms our granularity characterization really only applies to base use cases. Use cases can also be extended by other use cases and include other use cases, and these others will necessarily be of finer granularity than their base"[4]. Thus leaving some playroom for matching the level of detail of transactional function types to the granularity of use cases. What we have to be cautious about, however, is not to flood the UML Components model with use cases, making it hard to get a quick overview of the system functionality. In section 3.3.4 we describe how we deal with this.

2.4 Compatibility with IFPUG guidelines for FPA

The proposed solution to the problem in Chapter 1 is elaborated using the FPA guidelines from the NESMA user group, because those are the guidelines used at Capgemini BAS. The well known IFPUG (International Function Point User Group) guidelines for FPA [6] closely resemble the NESMA guidelines. In fact the IFPUG recognizes the same concepts as the NESMA. Both distinguish the same types of functions and complexity levels. Only some minor differences between the guidelines exist [13].
For example, the difference in the way of determining whether a transaction is an external inquiry or an external output is the fact that according to the NESMA the result of an external inquiry should be determined in size, otherwise it will be counted as external output. The IFPUG does not maintain this requirement for external inquiries. Another difference is the fact that the FPA tables are counted by the NESMA as one ILF and/or one EIF, while the IFPUG does not count any points for these tables, because the latter user group considers FPA tables as a way of implementing technical requirements, while only function require-

ments should be counted. Furthermore, there are a few differences due to the fact that some minor subjects are addressed by the NESMA guidelines while omitted by the IFPUG guidelines.
The slight differences between the guidelines have no big influence on the function point counts resulting from the guidelines. Also, these differences are not of such nature that these would have a great impact on the created mappings between the gathered information and the specification in UML if the IFPUG guidelines would be used instead of the NESMA guidelines. Therefore, it is also possible to apply this method with the IFPUG guidelines without having to implement major changes in the transformations presented in the next Chapter.

Chapter 3

Discovering mappings

In this chapter we will devise a case to explore the relationships between the entities that we are dealing with. First of all, we describe the various transformations that need to be implemented. Next, we elaborate a hotel reservation system, which will be used for the exploratory case study in which we determine the mappings that we need to know for the transformations. The reason for using a hotel reservation system is the fact that examples of such a system exist for both the UML components method and the NESMA standard. These examples can be molded into an example system which can be used for comparing the entities defined in the UML components method to the entities defined in the NESMA standard. Furthermore, we specify metamodels that are required for creating the models and we implement transformations which are used for automated mapping between the models.

3.1 Transformations and metamodels

First of all we should get a clear view on what needs to be created. Our goal is to perform automated mappings of information about a system by means of model transformations. We

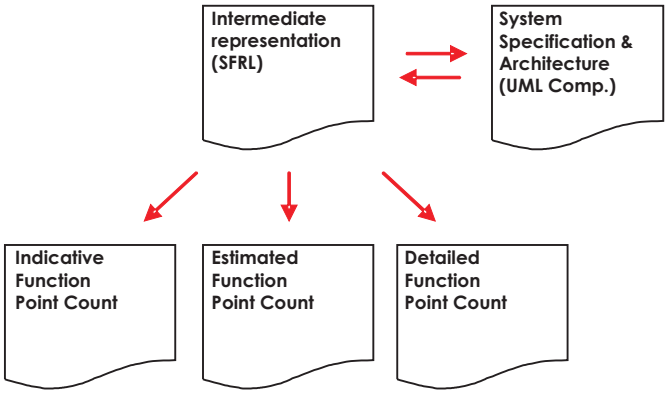

Figure 3.1: Transformations

need to ask ourselves what transformations we are going to implement. The next step is to see what more is required in order to create these transformations.

In the solution that we proposed in Chapter 1, we are dealing with several transformations (see Figure 3.1). We need to be able to perform function point counts based on the information recorded in SFRL. As we described in Chapter 2, the NESMA standard provides 3 different types of function point counts. For each of these types of function point counts a mapping should be created from SFRL to the function point count.

Also, a basic start of a UML components specification can be created based on the information recorded in SFRL. A transformation should be created which maps the information in SFRL onto information in a UML components specification. Furthermore, it should be possible to extract information from a UML components specification and translate this information into the SFRL format.

For each of these translation steps, model transformations have to be defined. A model transformation language aims to provide a means to specify the way to generate target models from a set of source models. A model is defined according to the semantics of a metamodel. A model that respects the semantics of a metamodel is said to conform to this metamodel. A metamodel itself is a model which has to conform to its own metamodel. Therefore, a third level model is defined, which is called a metametamodel (see Figure 3.2).

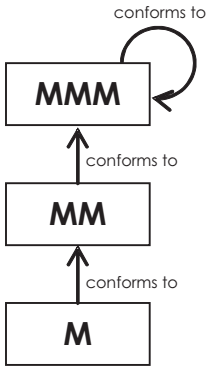

Figure 3.2: Model, metamodel and metametamodel

A metametamodel provides the semantics that are required to specify metamodels. Usually a metametamodel can be defined according to its own semantics. In that case a metametamodel conforms to itself.

A model transformation itself can be regarded as a model that conforms to a metamodel. A model transformation describes a mapping of certain metamodel onto another (or maybe the same) metamodel. When executing the transformation an input model conforming to the first metamodel is transformed into an output model conforming to the latter metamodel

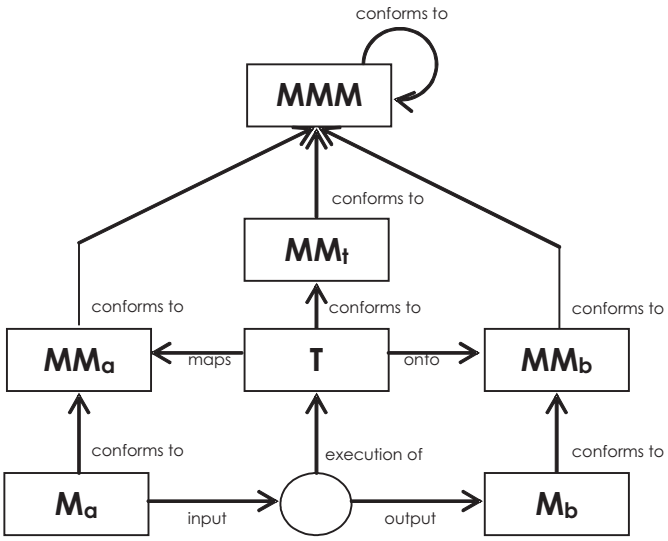

Figure 3.3: Model transformation

according to the mapping described in the model transformation (see Figure 3.3).

What we can say now is that we need to create transformations that map information from SFRL to either of the three types of function point counts (indicative, estimated or detailed function point count). Also, we need transformations that map information about a system in SFRL to a UML components specification of that system and vice versa.
In order to create these transformations we need metamodels for the various types of models involved. We need a metamodel according to which information can be recorded in SFRL (i.e. a metamodel that defines SFRL), a metamodel to which a UML components specification adheres, and a metamodel according to which the function point counts can be expressed.

3.1.1 Metamodel for a UML Components specification

First of all we will determine what metamodel is required for the UML components specification. For modeling a specification of the system according to the UML components method, UML is used. In our case we used UML 1.4, the metamodel of which is available online from the Object Management Group's website. The choice for using this version of UML for modeling the artifacts from the UML components method emanates from prior use in research in applying model transformations in the UML components method and the software used at that time. Although switching to UML 2.0 is possible by means of downloading and installing updates, there are no advantages involved by doing so. The UML components method does not use any UML model elements that are specific to version 2.0. Since the software environment was already set for use with UML 1.4 and because of the

gained experience with this version and the lack of advantages of upgrading to a newer version of UML the choice was made to continue using UML 1.4 for this research. However, it is very well possible to use UML 2.0 instead of UML 1.4 without complications by using an updated metamodel of UML and by fine tuning the model transformation to the new metamodel.

Cheesman and Daniels [4] described a profile for creating a UML components model. This profile exists of a package hierarchy and a description of what elements must be placed in which package and a description of the stereotypes that should be applied to those elements.
When transforming an SFRL model to a UML components model, not all information from SFRL can be given a place in the UML components profile as is. To be able to preserve certain information from SFRL when transforming to UML Components we slightly extended the UML components profile in our transformations. The additions made to this profile will be discussed in Section 3.3.4.

3.1.2 Metamodel for expressing Function point counts

Furthermore, we need a metamodel to express indicative, estimated and detailed function point counts in the form of models. To create models of the function point counts no metamodels existed yet. Therefore we needed to define a metamodel which specifies what a model of a function point count looks like. The main attribute that should be in the model is the amount of function points of which a system exists. The name of the system is a desirable attribute, otherwise we would have to remember which count belongs to which system, which we obviously don't want to do. Also, we chose to include in the model the number of occurrences of each type of function in order to give a quick indication of what the function point count is composed of (see Figure 3.4). As one can see we made a distinction between

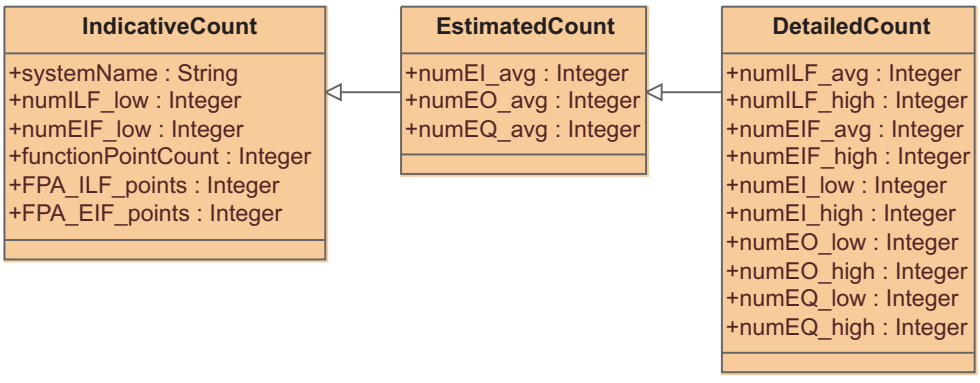

Figure 3.4: Metamodel for FPA

the three types of function point counts (indicative, estimated or detailed count). Furthermore, the metamodel could be made as comprehensive as desired, but to create an example for our research this will be sufficient.
The indicative count is the most simple type of count. In this type of count the number of Internal Logical Files (ILF) and External Input Files (EIF) is counted and each is assigned a low complexity. Furthermore, the amount of function points based on the presence or absence of FPA tables ILF and FPA tables EIF is recorded.
The estimated count is an extension of the indicative count. All elements recorded in an indicative count are also recorded in an estimated count. On top of that the number of external inputs (EI), external outputs (EO) and external inquiries (EQ) are recorded. Each type of these user transactions are assigned an average complexity.
The detailed count on its turn is an extension of the estimated count. For each type of file and each type of transaction the amount with a low complexity, the amount with an average complexity and the amount with a high complexity is recorded.

3.1.3 Metamodel for SFRL

In order to work with SFRL models in our transformations we need a metamodel for SFRL. For the first draft of SFRL, the elements from the NESMA standard were used, because the first objective is to capture the information gathered during the first analysis of the system in SFRL. Since the first analysis is done according to the NESMA standard, using the elements from this standard is a good starting point for the creation of SFRL. During this research the metamodel for SFRL evolved into the current metamodel.

After closely and repeatedly examining the NESMA standard a metamodel for SFRL evolved. First of all the system we are dealing with is recorded. According to the NESMA standard functions in a system can be determined. Two categories of functions are distinguished, data functions and transactional functions (see Figure 3.5).

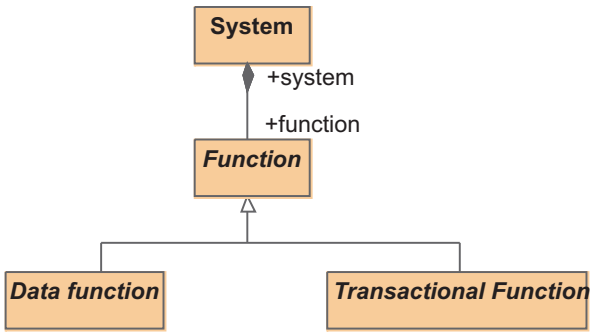

Figure 3.5: A system contains data functions and transactional functions

According to the NESMA specification various types of data functions exist. Data functions can be internal logical files (ILFs), external interface files (EIFs), FPA tables ILF or FPA tables EIF. The ILFs and EIFs are files which can consist of one or more record types (RETs). These RETs consist of data element types (DETs). The FPA tables also consist of data element types (see Figure 3.6).
Initially only the number of record types and data element types were recorded because these numbers are required to determine the complexity of the data types. However, to be able to count the records and data elements these have to be identified anyway. Whether we store each RET and each DET or only the number of RETs and DETs identified does not make a difference in the human effort required. Also, it can't harm to preserve information we already have, since it might be useful for more than only assigning complexities to files. Therefore, instead of just recording the number of record types and data elements, we choose to register each identified record type and each data element.

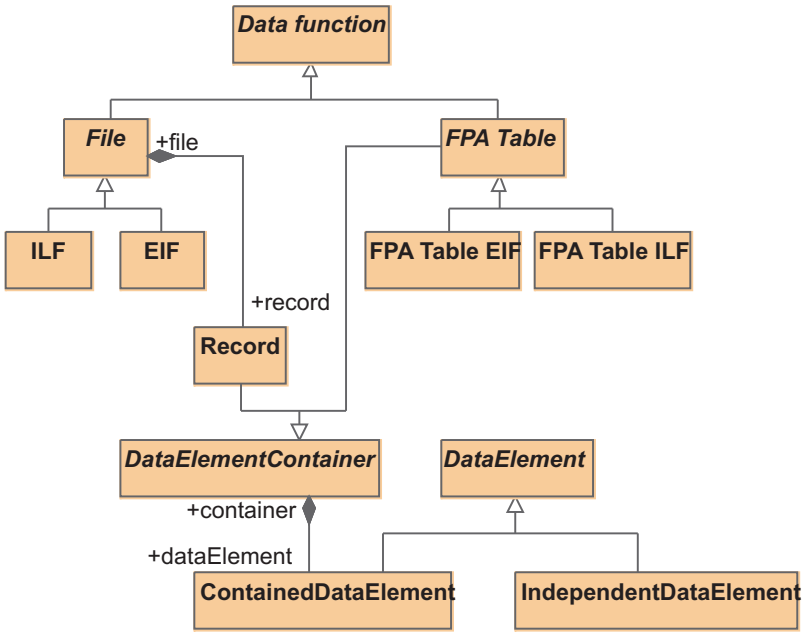

Figure 3.6: A depiction of data functions distinguished by the NESMA

Moving on to the transactional functions, the NESMA distinguishes the external input (EI), external output (EO) and external inquiry (EQ) functions. Transactions may keep references to data element types (DETs) crossing the system border which are used as input and to the ones which are used as output (see Figure 3.7).
To determine the complexity of transactions the number of DETs crossing the system boundary and the number of files referenced by the transaction are required. The number

of referenced files can be derived by identifying the DETs referenced by the transactions, after which the records which contain these DETs can be identified. Once we know which records contain the referenced DETs, we can easily count the number of files in which these records reside.

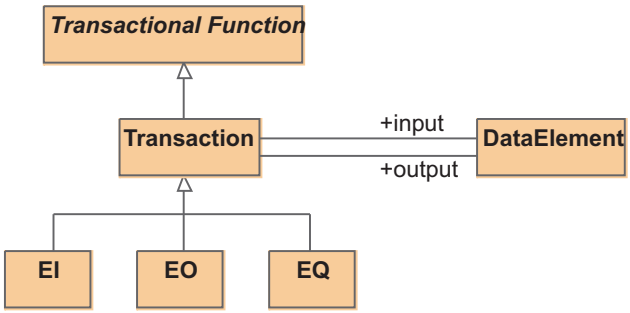

Figure 3.7: An initial depiction of transactional functions distinguished by the NESMA

We already explained why we chose to record all information we have about records and data elements in files when determining the complexity of these files. for the same reason we choose to preserve all information about DETs identified for determining the complexity of transactions. Instead of recording just the number of files referenced and DETs crossing the system boundary, we record the references to DETs. This may be DETs from data functions, which we will refer to as ContainedDataElements, as well as DETs from user screen elements or DETs derived from DETs in data functions, which in SFRL we will refer to as IndependentDataElements. The latter type of data elements are independent in the sense that they don't exist within a container. The connections between files and the records in these files are recorded as well as the connections between records and the data elements in these records. The number of referenced files can be derived from the references to DETs that are contained in records an thus in files.
For external inquiry functions either the input part or the output part, depending on which of the two has the highest complexity, determines the complexity of the total inquiry function.

Obviously, NESMA function point counts based on models created according a metamodel containing the information specified thus far can be performed without complications, because this metamodel would be based purely upon the NESMA standard. The next step is to map the recorded information to a UML components specification. During this step we had to deal with differences in detail levels between the information recorded according to the NESMA standard and the artifacts in the UML components method. Therefore, SFRL was adapted in such a way that it provided a means to distinguish elements on the level of detail of a function point count and to distinguish elements on the levels of detail used in the artifacts of UML components method.

What we need to bear in mind is that as much elements in the UML components specification as possible should be generated from the information available after the first analysis of the system. When a company performs a function point count for creating an offer while it is unknown whether it will get the order, that company does not want to spend much time in doing things that do not directly contribute to creating the offer. On the other hand, it is desirable to record the gained information in such a form that it can be used optimally once the company does get the order.

In Chapter 2 we explicated the distinction of use cases according to Cockburn [5] in goals at different levels. The reason for this is that this way of distinguishing the various levels of detail seems very applicable in our situation.
The goals described at the highest level, the summary level, can also be referred to as business goals. These goals describe the functionality within the borders of the system in an orderly way. What we will see in the example of the hotel system is that the summary level goals show us that the main purpose of the system is to maintain reservations, keep up with the room occupation and to manage information about the rooms that are for hire. The level of these goals, however, is too high to perform a function point count. The purpose of the system is clear, but the amount of functionality provided by the system to fulfill this purpose is not.
At a lower level we have the user goal level. This level shows a strong resemblance to the description of the desired granularity level of use cases as described by Cheesman and Daniels [4]. The distinction on this level is based upon the direct goals of a user of the system, such as making a reservation or adding a room type. According to Cheesman and Daniels the desired granularity level of use cases in a UML components specification should be based upon the direct goals of actors, which are in fact also the users of a system or external systems which use the functionality of the system. As we can see in Table 3.2,

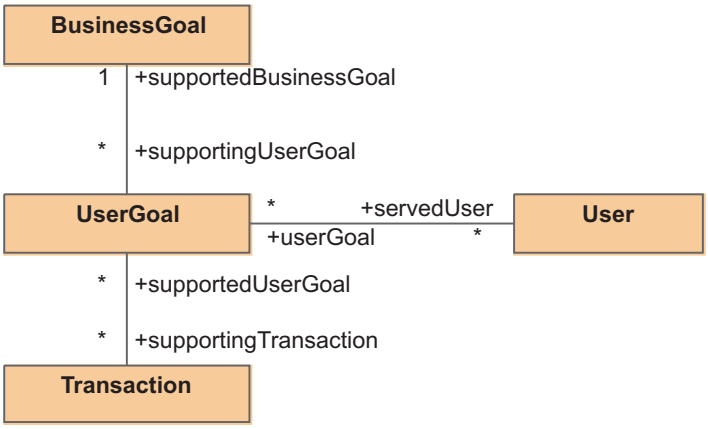

Figure 3.8: Introduction of different goal levels

the transactions in a function point count, however, may be slightly more detailed than the use cases in a UML components specification. For example, for adding a room only one transaction is sufficient, but for making a reservation multiple transactions are required. What we can conclude here is that the user goal level resembles the level of detail of use cases in a UML components specification, but is slightly too high for performing a function point count in most cases.
On the third and lowest level defined by Cockburn, the subfunctions of which user goals can exist are described. This level can be compared to the transactions in Table 3.2. Based on the goals described on this level a function point count can be performed.

In Figure 3.8 the relationship between the aforementioned levels and the users can be seen. Here we can see that the goals of a system can be recorded at summary level (business goals), user level (user goals) and subfunction level (transactions). Transactions can be used to reach user level goals, which on their turn contribute to a summary level goal. As we can see in Table 3.2, transactions can be used to fulfill multiple user level goals.

Depending on the completeness of the documentation the levels of the goals in the system can be determined during the first analysis. Determining these goal levels, however is not essential for a function point count. Only the transactions are required for that purpose. Therefore, the distinction of goals in different levels of detail is optional in SFRL.

When analyzing the system for determining the size according to the NESMA guidelines, no distinction is made between different actors. After all, for determining the size of the system there is no need to know which actors can use which functionality. The functionality needs to be present in the system and that is what counts.
However, when specifying the system according to the UML components method there *is* a

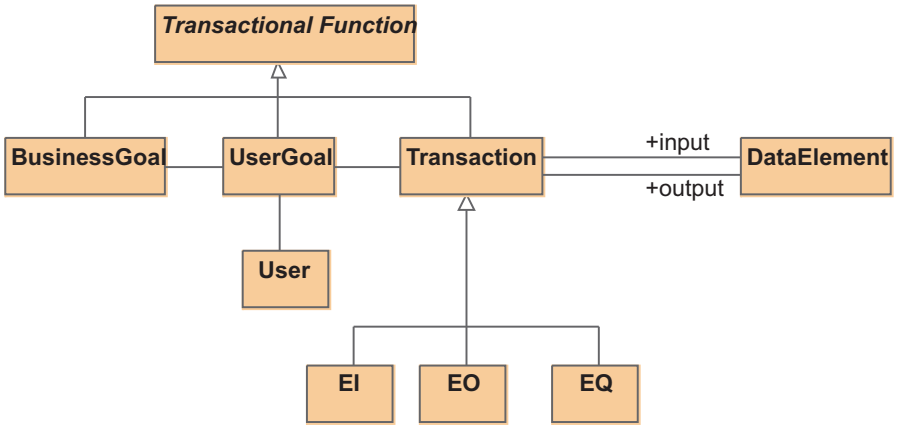

Figure 3.9: Transactional functions as represented in SFRL

distinction between various actors. If it is known which users should be able to reach which user goals, then it is desirable to have the option to record this information in SFRL. This way the information can be used when transforming the information into a UML components specification. A user may have multiple goals and, also, a user goal can be a goal of multiple users.

When combining the transactions as defined by the NESMA (Figure 3.7) with the users and goal levels (Figure 3.8) we get the transactional functions as represented in SFRL (see Figure 3.9).

If we put all this information together and properly name the references we get the metamodel for SFRL (see Figure 3.10).

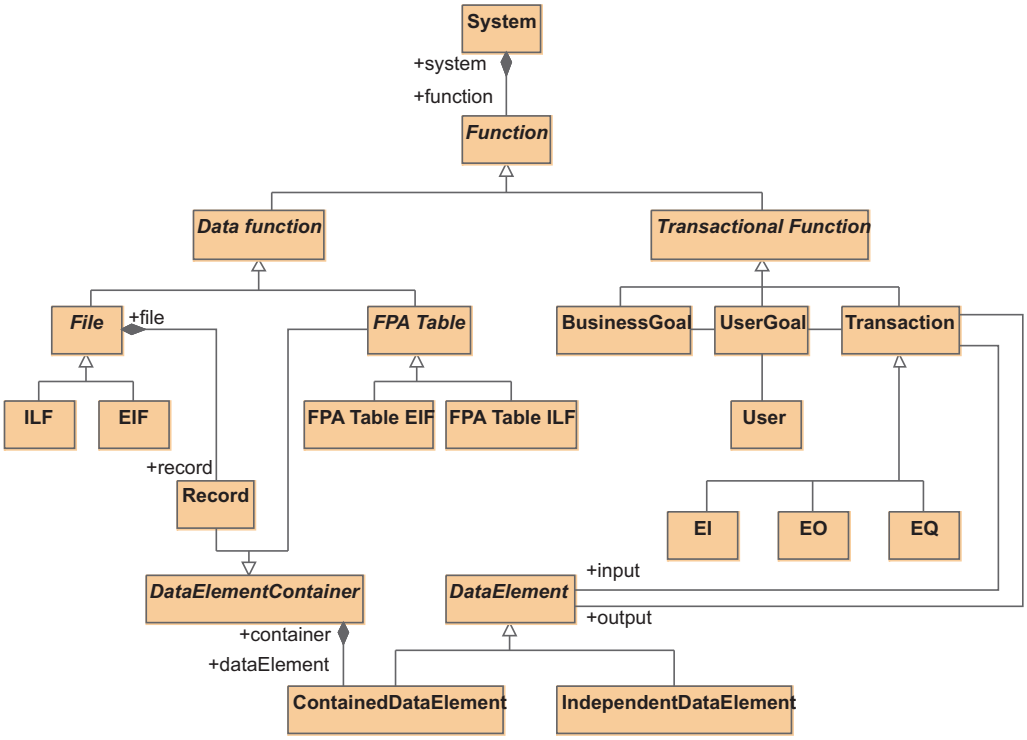

Figure 3.10: Metamodel for SFRL

The metamodels as defined in this section will be used in the transformations that need to be created. In these transformations rules are created which map elements in the source model's metamodel onto elements in target model's metamodel. The mappings will be defined later on in Section 3.3.

3.2 Exploratory case: Hotel Reservation System

In this section we describe an example case which was initially used for exploratory investigation. By examining this example we pointed out entities that can be mapped onto each other. The example system we used is a hotel reservation system. The hotel reservation system is frequently used as an example, because it is a small system, yet it is representative of systems that one might have to deal with. Both, the NESMA and the authors of the UML components method have used hotel reservation systems as examples to illustrate their techniques. The hotel reservation system example that we describe here is based on the hotel reservation system example as described by Cheesman and Daniels in their book on the UML components method [4] and the hotel case as provided by the NESMA [10]. Though, our system has been slightly adapted in order to meet our needs. Later on the example was used for testing the transformations.
In the NESMA version of the hotel reservation system a billing subsystem is included. The billing system is considered external and is therefore not elaborated in the UML components version of the hotel example. Since no elaborated UML components counterpart was available for the billing system, we also omitted the billing system from the function point analysis.
Furthermore, slight changes have been made in the way the functions are grouped together. In our example we tried to take the three goal levels as defined by Cockburn [5] into account when grouping the functions. The reason for doing so is because we integrated these goal levels into SFRL in order to meet the levels of granularity of both the use cases in the UML components method as well as the transactions in function point analysis. However, this is rather a notation question than a change in the example system.
Another change in the example was made in the way rooms and room types are managed. The reason for this is that before this change there was no example of a file with multiple records. After this change a room type is not just a simple attribute of a room, but is a manageable record. Room types can be added to, altered in or removed from the system.
Furthermore, very slight changes were made to make both examples more compatible. For example, in the UML components example functionality for checking the room availability is mentioned, while it is not counted in the NESMA example. In our example we added a transaction for checking the room availability.
In the next sections we will give a textual description of the requirements of the system, perform three types of function point counts and provide a use case model and a business concept model from the UML components specification of the system.

3.2.1 Requirements

The requirements of the hotel reservation system as specified in this chapter are based upon the requirements as specified in the NESMA hotel case [10]. Aside from the changes mentioned above, in our description of the system the data elements used by the system are mentioned, which in the NESMA case can be derived from screen information and by reading the detailed system description.

The main functionality of the hotel reservation system is to provide a means to support the renting of rooms.

In the hotel reservation system, room information (room number and room type) is maintained. To support this room types (name, price, description in English, Dutch, French and German) are mainained.

It should be possible to make a reservation for a certain period in time. First, the user makes a reservation request by filling in the required information: customer information (customer number, name, address, city, country code, telephone number, language, e-mail), the desired room type, number of rooms, date of arrival and the length of the stay in days. It is possible to add rooms of different room types to the reservation. The country code can be looked up and converted by utilizing a conversion table from an external system.
The system checks the availability of the number of rooms of the desired room types in the specified period of time. If the rooms are available in the specified period in time a confirmation is made in the appropriate language with the hotel details (hotel name, address, city, country, telephone number, fax number, e-mail, name of hotel manager) and the reservation details including a reservation number and the reservation date. If insufficient rooms of some selected room type are available, the system returns alternative room types from which the user can choose, after which a confirmation is made.

It should be possible to amend a reservation. The user fills in a reservation number or customer name, after which the reservation details are shown in a similar form like the one when first making the reservation in which the reservation details, except for the reservation number, can be changed. If multiple reservations with the same customer name exist, a selection screen appears for choosing the reservation that needs to be altered.

It should be possible to cancel a reservation. The user fills in a reservation number or customer name, after which the reservation details are shown on screen and a confirmation is asked. If the user confirms, the reservation is canceled. If multiple reservations with the same customer name exist, a selection screen appears for choosing the reservation that needs to be canceled.

When the customer arrives, he can take up a reservation. The customer provides his name or reservation number, which the hotel employee will use to lookup the reservation. If more than one results appear of which the date of arrival is equal to the current date, a selection screen appears for choosing the reservation that needs to be taken up. The reservation details are shown on screen. Here it's also possible to alter the customer details. The customer can perform a room registration for every room in the reservation. During this room registration the customer details, the room number, the arrival date, duration of the stay and room type are registered. For every registered room under a reservation, the number of registered rooms is updated. Once the number of registered rooms equals the number of reserved rooms, that reservation will be deleted.

When a customer arrives who has no reservation, he can register a room without making a reservation. The user chooses a room type in the room type selection screen and submits the desired duration of the stay in days. If a room of the selected room type is available during that period, the customer provides the customer details. The registration details (room type, room number, arrival date, duration of stay, customer details) are stored in the system.

When the customer departs from the hotel, he provides his room number to unregister. The current date is stored as departure date in the room registration. This information will later on be used by the billing system. Also, it's possible to request an overview of the costs of the stay. Information about this is retrieved from the billing system.

It should be possible to manage room information. By looking up a room number, the room details will be shown in a screen in which the room type can be altered. It should be possible to add or remove rooms.

Furthermore, it should be possible to manage room types. By looking up a room type, the details of a room type will be shown in a screen in which the price and descriptions in the four languages of a room type can be altered. It should be possible to add and remove room types. When removing a room type there may be no reservations or active room registrations for that room type. A confirmation is asked to the user before the room type will be removed.

3.2.2 Function point analysis

In this section we will identify the functions in the system described above. The overviews of the data functions and the transactional functions are based on the overviews of these functions in the NESMA hotel case [10]. Besides the fact that our tables are in English instead of Dutch, there are some other slight changes.
Due to the slight changes in the requirements the functions that belong to the billing subsystem are omitted. Instead a transaction for consulting the billing subsystem ("request cost overview") has been added.
Another change is the fact that in our system rooms and room types are managed separately. In the NESMA case the room type is a data element type of a room. A disadvantage of this approach is the fact that for listing the room types all the rooms in the system have to be checked for unique room types, while in our case the room types can easily be listed. Although, our goal is not to improve the hotel case. The reason behind this change is the fact that in the NESMA case no files with multiple records existed and now, due to this change, a file with multiple records occurs in the system.
Furthermore, we added function for checking whether a room is in use. This check was mentioned in the requirements of the NESMA hotel case, but strangely enough a function for this purpose was absent.

Indicative function point count

To perform an indicative function point count, the data function types have to be identified from the requirements specified in the previous section (see Table 3.1).

Function	***Type***
Reservation	ILF
Hotel	FPA table EIF*
Land	FPA table EIF*
Customer	ILF
Room Registration	ILF
Room & Room type	ILF
Bill	EIF
* All FPA tables EIF together count as one EIF	

Table 3.1: Data function types in the hotel reservation system

For each internal logical file (ILF) 35 points are counted, for each external input file (EIF) 15 points are counted. The FPA tables ILF and FPA tables EIF are alltogether counted as one ILF and one EIF respectively. If we perform an indicative function point count of the system the amount of function points counted is:

- 4 ILFs: 4x35 = 140 function points
- 1 EIF: 1x15 = 15 function points
- FPA tables EIF present: 15 function points

total: 170 function points

Estimated function point count

To perform an estimated function point count, besides the data function types identified above we also need to identify the transactional function types in the system (see Table 3.2).

Function	***Type***
Manage reservations	
Make reservation	
Request reservation	Input
Lookup room types	Output
continued on next page	

continued from previous page	
Check room availability	Output
Confirm reservation	Output
Alter reservation	
Lookup reservation	Output
Alter reservation	Input
Cancel reservation	
Lookup reservation	*
Cancel reservation	Input
Manage room occupancy	
Register room with reservation	
Lookup reservations arriving today	Output
Register with reservation	Input
Alter customer details	Input
Register room without reservation	
Register without reservation	Input
Lookup room types	*
Check room availability	*
Unregister room	
Unregister	Input
Request cost overview	Output
Manage rooms	
Add Room	
Add room	Input
Remove Room	
Lookup room	Inquiry
Check whether room is in use	Output
Remove room	Input
Alter Room	
Lookup room	*
Check whether room is in use	*
Alter room info	Input
Add Room type	
Add room type	Input
Remove Room type	
Lookup room type	Inquiry
Check whether room type is in use	Output
Remove room type	Input
Alter Room type	
continued on next page	

continued from previous page	
Lookup room type	*
Check whether room type is in use	*
Alter room type	Input
	** function has been counted before*

Table 3.2: Transactional function types in the hotel reservation system

This function point analysis is based on the NESMA FPA Case Hotel [10], but as one might notice, the transactions are grouped in a slightly different way. The reason for this is that now, the goal levels as defined by Cockburn [5] consistently come to expression in this table. The bold elements can be seen as summary level goals, the italic elements in the table can be seen as user level goals, while the actual transactional function types that are counted can be seen as subfunction level goals.

As we will see later on, this consequent way of grouping the transactional function types will help us in mapping these to UML. After all, if the detail level at which the functions are grouped would vary, then it would be extremely hard to set up rules that could make a connection between the detail levels in FPA and the detail levels in UML.

Each file will be regarded as a data function type of low complexity, resulting in 7 points for each ILF and 5 point for each EIF. The transactions will be considered of average complexity, resulting in 4 points for each Input, 5 point for each output and 4 points for each inquiry. If we perform an estimated function point count the amount of function points counted is:

- 4 ILFs: 4x7 = 28 function points
- 1 EIF: 1x5 = 5 function points
- FPA tables EIF present: 5 function points
- 13 EIs: 13x4 = 52 function points
- 8 EOs: 8x5 = 40 function points
- 2 EQ: 2x4 = 8 function points

total: 138 function points

Detailed function point count

To perform a detailed count, we have to determine the complexity of the files and transactions identified above. To determine the complexity of files the amount of record types in each file and the amount of data elements in those record types have to be determined. To determine the complexity of the transactions the amount of files that were referenced by the transaction and the amount of data elements that cross the system boundary have to be determined.

No detailed function point count was available in the NESMA case. Therefore, we had to perform the detailed function point count ourselves by carefully reading the detailed

system specification and by counting the number of data elements and records in each file and by counting the number of referenced data elements and the number of files in which these data elements exist. This detailed function point count was reviewed by a function point analyst at Capgemini BAS.

Complexity of files
All the files in the hotel reservation system are files with a low complexity. Details on the determination of the complexity of the files can be found in Appendix A.

Complexity of transactions
We have determined that the complexity of the transactions in the hotel reservation system is as follows:

Function	***Type***	***Complexity***
Request reservation	EI	average
Lookup room types	EO	low
Check room availability	EO	average
Confirm reservation	EO	high
Lookup reservation	EO	low
Alter reservation	EI	average
Cancel reservation	EI	average
Lookup reservations arriving today	EO	low
Register with reservation	EI	low
Alter customer details	EI	low
Register without reservation	EI	average
Unregister	EI	low
Request cost overview	EO	average
Add room	EI	low
Lookup room	EQ	low
Check whether room is in use	EO	low
Remove room	EI	low
Alter room info	EI	low
Add room type	EI	low
Lookup room type	EQ	low
Check whether room type is in use	EO	average
Remove room type	EI	low
Alter room type	EI	low

Table 3.3: Complexity of the transactional function types in the hotel reservation system

Details on the determination of the complexity levels of the transactions can be found in Appendix A.

Function point count
Now we know the complexity of each data function type and each transactional function type. Using Table 2.2 we can perform a detailed function point count. The amount of function points counted is:

- 4 ILFs of low complexity: 4x7 = 28 function points
- 1 EIF of low complexity: 1x5 = 5 function points
- 2 FPA tables EIF with 10 DETs: 5 function points
- 9 EIs: of low complexity: 9x3 = 27 function points
- 4 EOs: of low complexity: 4x4 = 16 function points
- 2 EQs: of low complexity: 2x3 = 6 function points
- 4 EIs: of average complexity: 4x4 = 16 function points
- 3 EOs: of average complexity: 3x5 = 15 function points
- 1 EOs: of high complexity: 1x7 = 7 function points

total: 125 function points

3.2.3 Creating an SFRL model after analysis of the system

After a function point analysis information about a system has emerged from which we can create an SFRL model according to the SFRL metamodel we created earlier on. Initially, the creation of SFRL models from the information emerging from analysis of the system was done in XMI using just a text editor. As one might understand this way of creating SFRL models is not desirable, because it is not user friendly and can be very time consuming. In Section 3.5 we come up with an alternative for inputting information into an SFRL model. In this section visual representations of the hotel reservation system information in SFRL can be found.

3.2.4 System specification

Generally, once a customer agrees with an offer made the system will be specified in UML, in our case according to the UML components method. In this section we will provide the business concept model and the use case model for the hotel reservation system created manually according to the UML components method.

We will start with the business concept model. It is a straightforward model in which the concepts a hotel reservation system has to deal with are modeled. This model is based on the business concept model created by Cheesman and Daniels, but has been slightly changed to comply with our example. The biggest change is the fact that room registration has been added to the model, which was absent in the example of Cheesman and Daniels.

In a use case model the functionality offered by the system to actors is modeled. This is

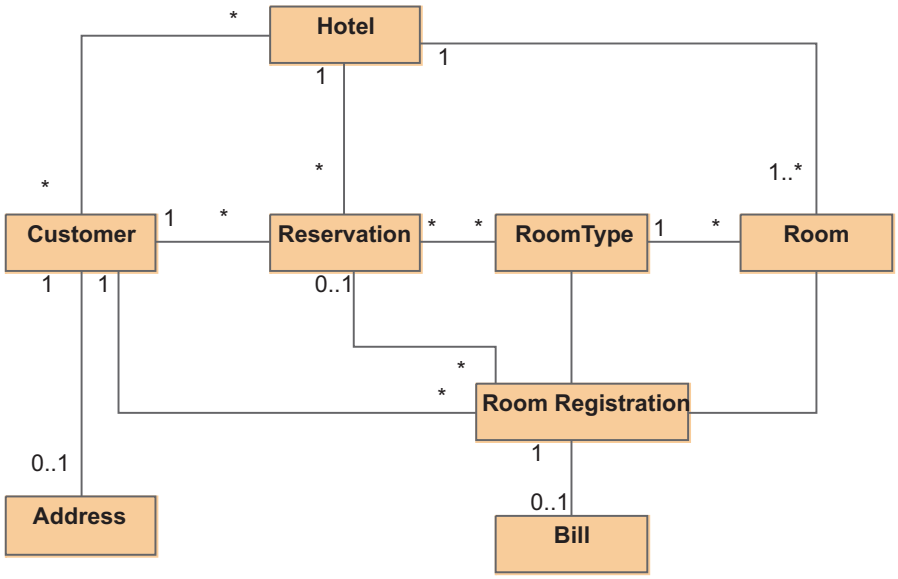

Figure 3.11: Hotel Reservation System, Business Concept Model

visualized by use cases, actors and the relationships between these elements. One thing we have to deal with, as mentioned in chapter 2, is determining the granularity of the use cases. Cheesman and Daniels [4] pointed out that a use case should be smaller than a business process but larger than a single operation on a single component and that the purpose of a use case is to meet the immediate goal of an actor. If we translate that to a goal level according Cockburn [5], that would mean Cheesman and Daniels suggest creating use case models at user goal level. Our use case model therefore contains use cases with the same names as the user goals in our example. This resembles the (not fully elaborated) use case model from the example of Cheesman and Daniels (make, alter, cancel reservation, etc.).

If one would create a use case model on the user goal level, it should look something like Figure 3.12

The business concept model can be further elaborated as a business type model. The use case model can be further elaborated as system interfaces and initial operations.

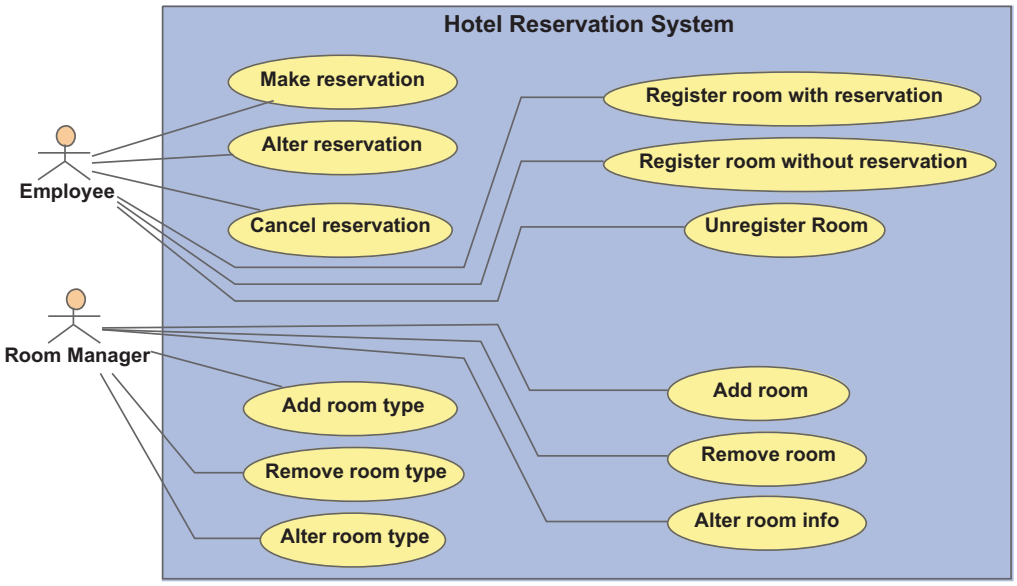

Figure 3.12: Hotel Reservation System, Use Case Model

3.3 Mappings

The function point counts and UML components models as specified above contain information that can be mapped onto each other. We use the example above to determine how model elements can be mapped. In this section we will describe how model elements should be mapped for the transformations and how these mappings have been determined.

3.3.1 From SFRL to indicative function point count

The most simple of the mappings is the transformation of information in SFRL into an indicative function point count. When performing an indicative count the number of internal files recorded in SFRL is multiplied by 35 when using a conceptual data model. When using a normalized data model, the number of internal files is multiplied by 25. The number of external files is multiplied by 15 when using a conceptual data model and 10 when using a normalized data model. If one or more occurrences of FPA tables ILF and/or FPA tables EIF exist in the system, these are counted all together as one extra ILF and/or one extra EIF.

At Capgemini BAS a tool is used for recording function point analyses. The creators of the tool assume that information recorded in the tool is based on conceptual data models. When generating a function point count from information recorded in SFRL we will make the same assuption. It is very easy however, to alter the transformation for use with normalized models if desired.

In Section 3.1 we defined metamodels for SFRL and for function point counts. To create a transformation that generates an indicative function point count we need to define a rule that maps elements from the SFRL metamodel onto an IndicativeCount element from the function point count metamodel. We want to perform a function point count of a system, therefore we map the System model element from the SFRL metamodel onto the IndicativeCount element. The values of the attributes can easily be set. The attribute systemName can simply be copied from the attribute name of the element System. The number of occurrences of each type of file can be set by counting the number of function references in the System in which a model element referenced is a File. The number of function points that need to be assigned as a result of the presence of FPA tables is set by checking whether function references in the System exist in which the referenced model element is an FPATable. If no references exist, no function points are assigned. Eventually, all function points are added up and the total number of function points is stored in the output model in the attribute functionpointCount.

The pseudocode for this mapping can be found in Appendix B Section B.1.

3.3.2 From SFRL to estimated function point count

This transformation is not more complicated than the previous one. The difference is that a System element from SFRL is now mapped to an EstimatedCount element in the function point count metamodel. The EstimatedCount element is a minimal extension to the IndicativeCount element. The amount of function points assigned to files is lower and in this type of count also the transactions and their type are identified and function points are assigned to them. The transactions are treated as transactions with an average complexity.

The pseudocode for this mapping can be found in Appendix B Section B.2.

3.3.3 From SFRL to detailed function point count

To realize this transformation the complexity of every file and every transaction has to be determined. In SFRL it is possible to record information about the system in such detail that it can be used for determining the complexity of transactions and files in the system. When sufficient details are recorded for doing so, we want to use the information for performing a detailed function point count. At the start of a project the documentation hardly ever contains enough information for a detailed function point count. For completeness we will define this transformation anyway. However, for it to be useful the detailedness of the supplied information about a system needs to increase.

In SFRL references from transactions to the data element types used can be recorded. This way the amount of data element types used can be derived. Also the amount of referenced files can be derived from this information. Based on the amount of data element types used and the amount of referenced files the complexity of transactions can be determined using Tables 2.5 and 2.6. In table 2.2 the amount of function points that will be asigned can be looked up.

For each file the amount of record types in the file can be recorded. For each record type the amount of data elements in it can be recorded. Based on the amount of record types and the total amount of data element types in a file the complexity of that file can be determined using Tables 2.3 and 2.4. Again, in Table 2.2 the amount of function points that will be assigned to the file can be looked up.

In the hotel case for example we can see that the rooms and room types file contains two record types. For each room the room number and room type can be recorded. For each room type the name of the type, a price, and a description in four languages can be recorded. So in total there are eight data element types in the file. Even though the file contains two record this still is a simple internal logical file, because of the low number of data element types.

This is also a relatively simple transformation. Because of the form of SFRL the required information can be recorded in such a way that no complex operations have to be performed to gather the information for the target model.

The pseudocode for this mapping can be found in Appendix B Section B.3.

3.3.4 From SFRL to UML

During this transformation data about the system recorded in SFRL needs to be transformed into elements in a UML components specification. This mapping is more interesting than the previous ones in the sense that decisions about how to map these elements need to be taken. In the previous transformations only one way of mapping can be used when following the NESMA guidelines, while for this transformation no guidelines exist on how to map the elements. We chose mapping solutions which we found suitable, however, possibly multiple solutions can be thought of. In that case, of course, it is possible for the user to choose another solution and alter the transformation to that choice. What we want to show here is the principle.

An example of a decision that needed to be taken is how goals and transactions on the one side of the transformation and use cases on the other side of the transformation relate to each other. In Section 3.1.3 we already argued that user goals in principle correspond to use cases in the UML components method. During the analysis of the system, however, goals can be identified at different detail levels. When performing an estimated count, the identification of goals should obviously be done at subfunction (transaction) level.

We could choose to position the transactions as use case steps within use cases deduced from user goals. Another option is to create use cases for each transaction as well. We can then make dependencies with the stereotype ≪include≫ between use cases deduced from user goals and the use cases deduced from transactions that support these user goals.

The first option would mean that a transaction could not be specified any further into smaller steps. According to the NESMA standard [11] a transaction must be an elementary function. As a result of this, it might seem as if a transaction can not be divided into smaller steps. If we look closer we will find out that this is not true. For a function to be elementary it has to satisfy two conditions. *The function has an autonomous meaning to the user and fully*

executes one complete processing of information and *after the function has been executed, the application is in a consistent state* [11]. As an example entering a new employee in a system is given. From the user's perspective this is one single process of which entering the new employee's name and entering the new employee's salary data is a part. Also, when we look at the confirm reservation use case for example, we notice that this use case could be split up in the following steps: determine the customer language, retrieve reservation data, retrieve hotel data, create and send a confirmation in the appropriate language. If we would position the transactions as steps in the use case, we would not be able to further specify smaller steps within these.

Also, what we have to bear in mind is that the identification of user goals is optional. Another objection against recording transactions as use case steps within a use case at user goal level is the fact that when no user goals have been specified in SFRL, no use cases will be generated in which the transactions can be positioned as use case steps. Since use case steps can only be placed within use cases, it would simply not be possible to create a counterpart for transactions in UML and the information in transactions would be futile. This is not acceptable, because often only the transaction level is identified after a function point analysis.

With the second option we don't have these problems. Whether the user goal level is identified or not, we can create counterparts for transactions in UML and if we wish we can even specify the details of a transaction. In the UML components method the authors already pointed out the possibility of including use cases in other use cases using ≪include≫ relations. So the ≪include≫ stereotype already exists within the UML components profile. We create use cases for every user goal and create ≪include≫ dependencies to all use cases created from transactions supporting the user goal. This way of mapping the information directly supports the fact that transactions can contribute to multiple user goals, as we can see in Table 3.2. In the same way we create use cases from business goals and create dependencies to use cases from user goals which support the business goal.

To maintain the distinction between the use cases generated from the different goal levels, we put the use cases in different namespaces. Since user goals have the desired detail level for the UML components method, use cases deduces from user goals will be put directly in the Use Case Model Package of the UML components model. Within this Package we create two additional packages. One for the use cases derived from business goals and one for use cases derived from transactions. To maintain the distinction between the various types of transactions we assign stereotypes to the use cases generated from these transactions. These stereotypes are named according to the type of transaction we are dealing with (i.e. ≪input≫ for external inputs, ≪output≫ for external outputs and ≪inquiry≫ for external inquiries).

Another advantage of separating the use cases at different detail levels is that the use case diagram will get far less crowded. If we want to get a quick overview of the business goals of a system we turn to the business goal package. If we want to get a quick overview of the functionality provided to the users we turn to the use cases at user goal level. If we want the full details we turn to the use cases in the Transactions package.

In SFRL it is also possible to record users which may have associations with user goals. Users recorded in SFRL can be translated to actors in UML with associations to use cases

generated from user goals (see Figure 3.13). Note that text between square brackets in names of elements represents variable text that depends on the properties of elements in the source model.

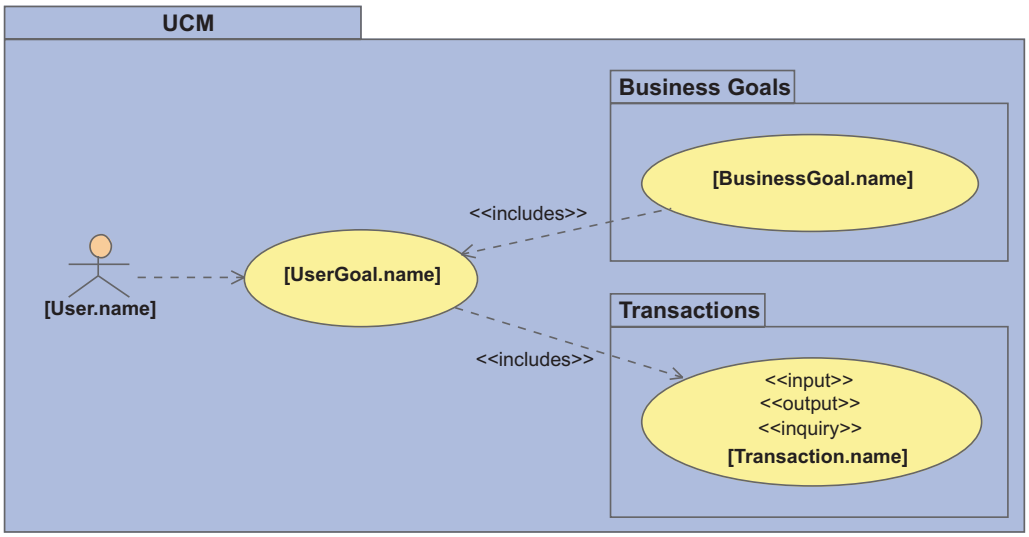

Figure 3.13: UCM diagram built from SFRL information

Furthermore, we need to determine how files should be translated to concepts in the business concept model. Our first thought is to map all files to business concepts. However, if we look at the hotel reservation example system, we see that address from the business concept model in Figure 3.11 does not occur as a file in the function point analysis. This is because the address is only used by customer and is included as an attribute of customer. However, this does not have to be a problem. If we look at the hotel reservation example elaborated by Cheesman and Daniels, we notice that the same happens in their case, only in a later stage. When refining the business concept model, address is absorbed by customer and becomes an attribute of it. What we might say is that mapping files to concepts leads to a refined version of the initial business concept model in Figure 3.11.

Furthermore, we see that land does not occur as a business concept in this business concept model. When automatically mapping the files to concepts, land would be represented as a concept in the model while in the hotel reservation case of Cheesman and Daniels a concept address instead of a concept land is created. Land and address both represent some notion of location, so what we can say is that the creation of model elements according to this mapping would not lead to an absurd result.

What comes to attention is the fact that in the manually created business concept model a concept room and a concept room type exist. If we compare this to how these elements are

manifested in SFRL, we see that room and room type are record types in a file. The other files contain only one record type each. What we can say is that by mapping the records from files to concepts instead of mapping the files themselves to concepts a business concept model is generated which is a closer match to the manually created business concept model.

What we need to keep in mind is that there might be projects in which the files are specified in SFRL but no record types are identified and recorded. In that case we will map the files to concepts in order to be able to generate a business concept. By doing so a business concept model will be generated which comes close to the business concept model in which concept are derived from records, since in most cases files contain only one record type.

However, it is desirable to make a distinction between these two ways of creating a business concept model, because at a certain point in time we may want to go back from a UML model to SFRL. If no distinction would be made it is hard to determine whether the concepts should be translated to files or to record types of files. Therefore we chose to create the mappings in the follow manner. In a UML components model a package Business Concept Model is created inside the package Requirements. The business concept model will be generated inside this package. Files are translated to classes with the stereotype ≪FPA File≫. If record types are present in SFRL, these will be mapped onto inner classes of the classes derived from files. These inner classes will carry the stereotype ≪concept≫.

Furthermore, in the NESMA standard FPA Tables are distinguished from ”normal” files. FPA tables can be considered as very simple files which have to meet certain conditions before they can be called FPA tables. These conditions are not relevant for the creation of a UML components specification of a system and therefore this distinction can normally not be made in the UML components method. Even though this distinction is not relevant for specifying the system, we still want to be able to maintain this information in order to be able to perform a function point count according to the NESMA standard later on when we extract information from the UML components model. Not being able to distinguish FPA tables from files can have a significant effect on the result of a function point count. Especially in case of an indicative count in which a considerable amount of function points is counted for each file, while the FPA tables are counted all together as one file. If all FPA tables would be counted as files, too many function points would be counted.

However, because of the way files and records are mapped to the business concept model we will be able to make a distinction between files and FPA tables. FPA tables can not contain multiple records are therefore be mapped to classes directly in the Business Concept Model package. These classes will also carry the stereotype ≪concept≫ (see Business Concept Model package in Figure 3.14). This way we can make a distinction between files and FPA tables (respectively classes directly in the business concept model package with the ≪FPA File≫ stereotype or the ≪concept≫ stereotype) and we can recognize which record types belong to which files. This will be useful later on when extracting an SFRL model from a UML components model.

Furthermore, we are dealing with files for which the system is responsible as well as files which are managed by external systems. As we stated in Section 2.3, in the UML Compo-

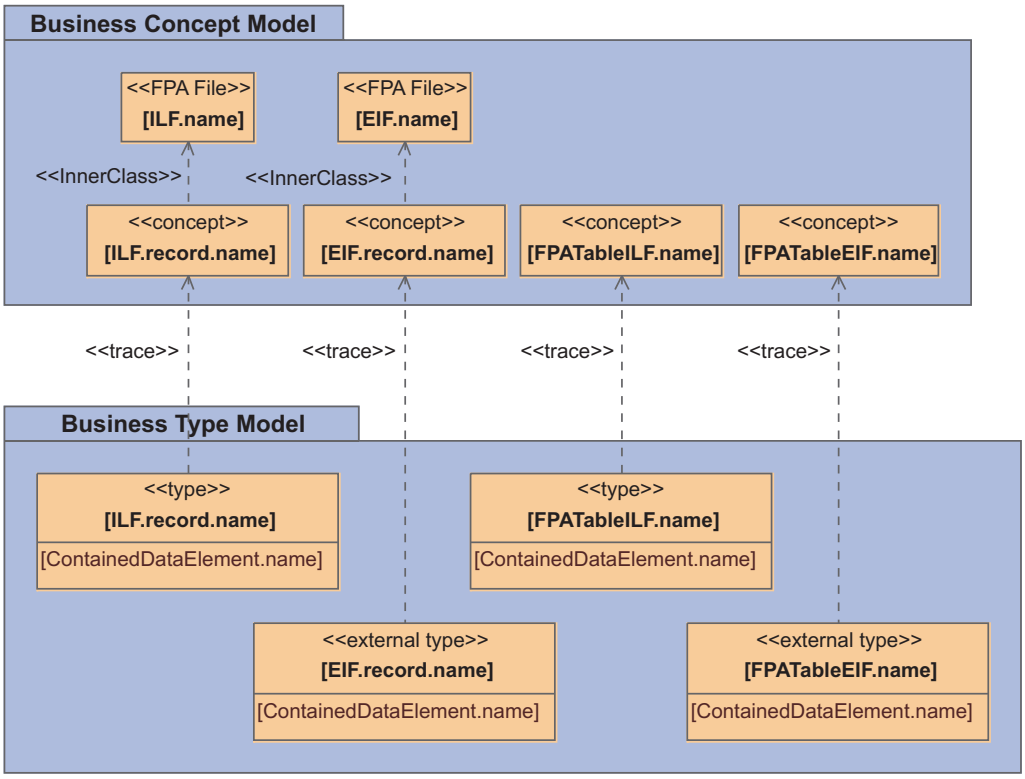

Figure 3.14: Business concept built from SFRL information

nent method the distinction between internally managed data and externally managed data can be found in the difference between the business concept model and the business type model. The first step in the creation of the business type model is scoping the business concept model by determining which concepts should be the responsibility of the system. The internal data (records in internal logical files and FPA tables ILF) are mapped to ≪type≫ classes in the business type model (see Business Type Model package in Figure 3.14).

The next step in the creation of the business type model is to add attributes to the types in the model. If the data elements of a record or FPA table are known and recorded in SFRL, these can be mapped to attributes of the type corresponding to that record or FPA table.

However, data elements from records in external interface files and FPA tables EIF will get lost, because these external data is not represented in the business type model. In order to preserve the data elements we will map the records from external interface files and FPA tables EIF to classes in the business type model anyway. To make clear that the maintenance of these functions is not responsibility of the system we will mark them with the stereotype ≪external type≫. Data elements which are neither part of any record nor of any FPA table are mapped to interface data types, which have the stereotype ≪data type≫.

Of course, we carefully considered whether looking at the difference between the business concept model and the business type model is the correct way of making a distinction between internal and external data. From the collection of types in the business type model the core types are identified. For each core type a business interface is defined which is linked to a business component. The functionality of the component is provided to the rest of the system or to external systems through the business interface. The data in the type is accessed by means of the functionality offered by this interface.

If we consider the hotel case used by Cheesman and Daniels [4], it can be noted that the concept bill is scoped away in the business type model, because managing bill is not the responsibility of the hotel reservation system. The data in bill is approached through a business interface of by the billing system. That business interface was derived from a type in the business type model of the billing system. In other words bill is a type managed by the billing system. This can be seen in the component architecture diagram of the hotel case used by Cheesman and Daniels. So external files are accessed through interfaces provided by external systems and can not be directly accessed by internal interfaces. Therefore only internal data should be represented as types in the business type model of the system that is modeled. External data is represented as a type in the business type model of an external system. So displaying external data as external types is a good option.

What we can conclude now, is that this way of making a distinction between internal files and external files is correct. All data functions are mapped to concepts in the business concept model. All records from internal files are mapped as business types in the business type model. When the UML components models are further elaborated manually, some of the types in the business type model can be identified as core business types. For these core business types business interfaces are created. These business interfaces can not be created by our transformations, since we can not identify the core types automatically.

When looking at the UML Components hotel reservation system example described by Cheesman and Daniels, we see that aside from the business interface created for these core types, business interfaces from external systems are also shown in the component architecture diagram. The system may use external data, for example billing data from the billing system. The business interfaces from external systems provide access to this external data. In a function point analysis external data is also identified in the form of EIFs. To access this data external business interfaces have to be created. Therefore, external files are mapped to interface types and will be shown in the component architecture diagram.

Within the UML components method system interfaces are derived from use cases, which on their turn are derived from user goals in SFRL. The operations in these use cases are derived from activities taking place inside the use case descriptions. These activities are at the same level of detail as described by Cockburn as the transactions in SFRL, namely the subfunction goal level.

Since the use cases from which the system interface are derived, on their turn are derived from user goals, we might also create a direct mapping from user goals to system interfaces. First a package is created for each system interface. Within this package a class is created with the stereotype ≪interface type≫. Subsequently we can map the transactions linked to the user goals to operations inside the interfaces. To preserve the knowledge about the

type of transaction (external input, external output or external inquiry), we give the operations stereotypes corresponding to the type of the transaction (≪input≫, ≪output≫ or ≪inquiry≫).

In UML Components info types are created for each system interface. An info type is in fact a refinement of a business type in which only the attributes are used in the system interface to which the info type belongs are contained. Each system interface is derived from a user goal. Usergoals are supported by transactions. The transactions may have input references and output references to data elements. These data elements are either contained by a DataElementContainer (record or FPA table) or exist independently. As we saw in Figure 3.14, a DataElementcontainer is the kind of SFRL model element that is mapped to a business type. By following these connections the DataElementContainers that are referenced by user goals, and thus the info types that have to be created for each system interface can be derived (see Figure 3.15).

Furthermore, the operations in the system interface contain parameters. In the UML component methods data value objects are used for exchanging information with operations. Since we do not have a way to derive the data value objects from SFRL, we could use the generated info types as an alternative. Another option is to create a parameter for each data element referenced by a transaction, but that might lead to a very large number of parameters in the operations. Therefore, we choose the first option. Also dependencies are created between the parameters and the attributes in the info type that are referenced in the operation.

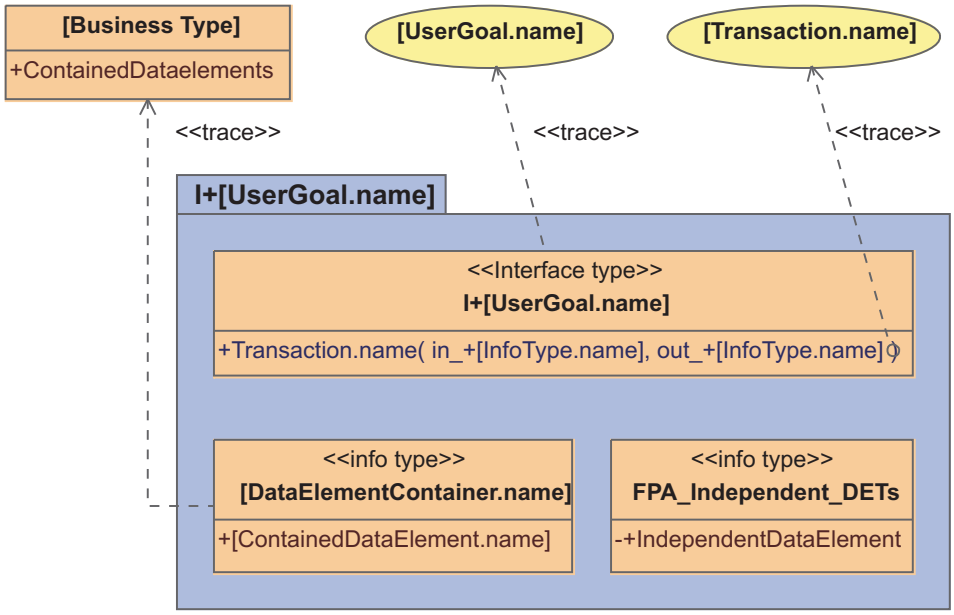

Figure 3.15: System interface built from SFRL information

The next step is to determine whether business interfaces can derived from the information we have. In the hotel case used by Cheesman and Daniels, customer and hotel are the core types. For each core type a business interface is created. These business interface provides functionality to manage the core type and the types immediately depending on the core type. If a business type immediately depends on multiple core types a decision has to be made on which business interface will be responsible for that type. For example, in their case the hotel business interface is responsible for the hotel, reservation, room and room type business types. Even though the reservation business type is also dependent on the customer core business type, using common sense the authors decided that reservations should be managed by the hotel business interface. This can simply be done by asking the question who is responsible for managing reservations, customers or the hotel? We can answer this question by looking at the real-life situation. We don't see hotel employees phoning customers to ask them details about reservations they made. On the contrary, if a customer wants to change something to a reservation they have to contact the hotel in order to do so. So it is the responsibility of the hotel to maintain reservations.

Since we mainly followed the NESMA case, in our case the situation is slightly different than in the case of Cheesman and Daniels. In the case of Cheesman and Daniels the hotel information is managed by the reservation system. Our reservation system retrieves the hotel information from an external system. The hotel details are not managed by the reservation subsystem itself. If the hotel type is managed by an external system there is no way that it can be a core business type of our system. After all hotel does not play a role in our business type model, or at the most it is an external type. If we compare our system to the hotel reservation system of Cheesman and Daniels we say that the core types can not be the same.

Initially we were wondering whether we might derive business interfaces from business goals, because coincidentally all our business goals are about managing some sort of data. What we have to keep in mind however is that when we look at FPA and also when we look at the UML components method, is that on the one hand we have the data side (for example files in FPA, concepts in the UML components method) and action side (transactions in FPA, use cases in the UML components method). Business interfaces are derived from core types, which are on the data side in the UML components method. Business goals are transactional functions in SFRL, which are on the action side in SFRL. We would very much like to say that business goals seem to provide a nice granularity for creating business interfaces, but would only be because we would just love to create business interfaces automatically. That is not the way business interfaces are realized in the UML components method. So based on the information we have in SFRL we can not create business interfaces.

Further elaboration of the UML components models, such as identifying core types and creating component specifications and a component architecture diagram will have to be done by the system specifiers. The difference for these system specifiers is that in the current situation the UML components models are created from scratch, while our approach creates a stepping stone based on information about a system we already have.

All the mappings as discussed in this section are summarized in Table 3.4. Furthermore,

pseudocode for the mapping can be found in Appendix B Section B.4. Once again we would like to point out that text between square brackets in names of elements represents variable text that depends on the properties of elements in the source model.

SFRL element	***Target model element***	***Name***
System	Model	[System.name]
EI	UseCase in /Requirements/Use Case Model /Transactions	≪input≫ [EI.name]
	Operation in System Interface derived from a supported usergoal	≪input≫ [EI.name]
EO	UseCase in /Requirements/Use Case Model /Transactions	≪output≫ [EO.name]
	Operation in System Interface derived from a supported usergoal	≪output≫ [EO.name]
EQ	UseCase in /Requirements/Use Case Model /Transactions	≪inquiry≫ [EQ.name]
	Operation in System Interface derived from a supported usergoal	≪inquiry≫ [EQ.name]
User	Actor in /Requirements/Use Case Model	[User.name]
UserGoal	UseCase in /Requirements/Use Case Model	[UserGoal.name]
	Depencency between UC's derived from user-goal and UC's derived from supporting transactions	≪includes≫
	Package in /Specification/Interface Specification /System Interfaces	I+[UserGoal.name]
	Class in /Specification/Interface Specification /System Interfaces/I+[UserGoal.name]	≪interface type≫ I+[UserGoal.name]
	Dependency between system interface and use case derived from user goal	≪trace≫
DataElement-Container referenced by Transaction	Class in packages of system interfaces which use the DataElementContainers	≪info type≫ [DataEementContainer .name]
	Parameter of Operation derived from Transaction using the DataElementContainer as input	in_+[DataElement-Container.name]
	Parameter of Operation derived from Transaction using the DataElementContainer as output	out_+[DataElement-Container.name]
	Dependencies between Parameters and attributes in info types that are use by the Operation	
BusinessGoal	UseCase in /Requirements/Use Case Model /Business Goals	[BusinessGoal.name]
ILF	Class in /Requirements/Business Concept Model	≪FPA File≫

continued on next page

continued from previous page		
		[ILF.name]
ILF.record	InnerClass of Class derived from ILF	≪concept≫ [Record.name]
	Class in /Specification/Business Type Model	≪type≫ [Record.name]
	Dependency between concept and type	≪trace≫
EIF	Class in /Requirements/Business Concept Model	≪FPA File≫ [EIF.name]
EIF.record	InnerClass of Class derived from EIF	≪concept≫ [Record.name]
	Class in /Specification/Business Type Model	≪external type≫ [Record.name]
	Dependency between concept and external type	≪trace≫
FPATable ILF	Class in /Requirements/Business Concept Model	≪concept≫ [FPATableILF.name]
	Class in /Specification/Business Type Model	≪type≫ [FPATableILF.name]
FPATable EIF	Class in /Requirements/Business Concept Model	≪concept≫ [FPATableEIF.name]
	Class in /Requirements/Business Concept Model	≪external type≫ [FPATableEIF.name]
Contained-DataElement	Attribute of a type in Business Type Model	[DET.name]
	Attribute of info types in system interface specifications	[DET.name]
Independent-DataElement	Class in system interface specification in which the data element is used	≪info type≫ FPA_Independent_DETS
	Attributes in FPA_Independent_DETS classes	[DET.name]

Table 3.4: SFRL to UML mappings summary

To sum up, the following changes were made to the standard UML components profile:

- Two packages were added: One for use cases at summary level (business goals) and one for use cases at subfunction level (transactions).
- The stereotypes ≪input≫, ≪output≫ and ≪inquiry≫ were added to be able to recognize from what kind of transaction use cases at subfunction level are derived.
- The stereotype ≪FPA File≫ was added to be able to distinguish between a class in the business concept model which is an FPA file and a class which is a concept.
- The stereotype ≪external type≫ was added in order to be able to add types derives from external files to the business type model without ruining the scope of the system. By adding external types to the business type model, information about data elements in external files is preserved.

3.3.5 From UML to SFRL

In this section we describe how the elements from a UML components model are mapped to SFRL. Depending on the information that can be extracted from the UML components models, we can perform a recount of the function points. What we are trying to do here is to reverse the mappings in the previous section. Based on the decisions taken and the conclusions made while mapping SFRL to UML Components diagrams, we will now see how the information processed in the UML models can be converted to information in SFRL. We assume that the UML components model from which information will be extracted is consistent. What we should keep in mind, though, is that a UML components model generated from an SFRL model may differ from a manually created standard UML component model. When creating the mappings from UML to SFRL we want to take both kinds of models into account.

To begin with, we will look at the data side of the system. As we saw in the previous section all files are mapped to classes with the stereotype ≪FPA File≫ in the business concept model. All records from the files are mapped to inner classes of these classes. These inner classes carry the stereotype ≪concept≫. FPA tables are mapped to classes in the business concept model and also carry the stereotype ≪concept≫.

When strictly following the UML components method all classes in the business concept model are concepts. It is possible to apply the stereotype ≪concept≫ to these classes, but this is not mandatory.

So in the business concept model there is likely to be a difference between a manually created model and a model generated from SFRL. An indicator for the kind of model we are dealing with is the presence of the stereotype ≪FPA File≫.

When this stereotype is present, the mappings are done in the following way. The classes in the business concept model with the stereotype ≪concept≫ are mapped to FPA tables. The classes with the stereotype ≪FPA File≫ are mapped to files. The ≪concept≫ classes within these classes are mapped to records in the files.

When the stereotype ≪FPA File≫ is not present in the UML components model we can not distinguish between files and FPA Tables. In that case all concepts are mapped to a file with a record with the same name as the file. The result will be less accurate than if we could make a distinction between files and FPA tables, but it will still be a good approximation since most data functions are files and most files contain one record.

Files and FPA tables can be managed internally or by an external system. By means of ≪trace≫ dependencies between types in the business type model and concepts we can determine which types are derived from which concepts. If a concept is linked to an external type, or to no type when strictly following the UML components method, we are dealing with an external file or FPA table. Otherwise we are dealing with an internal file or FPA table.

The data elements in the records and FPA tables are derived from the attributes of the types linked to the concepts from which the records and FPA tables are derived. Independent data elements are derived from the attributes of info types named 'FPA_Independent_DETs'

in the system interface packages or attribute from info type classes which do not occur in the corresponding business type classes.

Next, we will look at the 'action' side of the system. On this side we have the business goals, the user goals, the transactions and the users.

As we saw in the previous section the business goals are mapped to use cases in the Business Goal package in the Use Case Model package. We can simply reverse this process by mapping these use cases to business goals, that is if they exist in the UML components model. The ≪includes≫ dependencies between these use cases and use cases in the Use Case Model package tell us which business goals are supported by which user goals.

The next step is to see how we can retrieve the user goals from the UML components model. As we have seen in the previous section the user goals are mapped to use cases and to system interfaces. Transactions supporting the user goals are mapped to operations inside the system interfaces.

So what we can say is that user goals can either be derived from use cases or from system interfaces. System interfaces are derived from use cases, so in a UML components model in which system interfaces are present the use cases are very likely to be present too. However, the presence of use cases does not automatically mean that these have already been refined into system interfaces. Therefore, we choose to map use cases in the Use Case Model package to user goals. Note that the situation in which use cases are present while system interfaces are not, will only occur if the used UML components model was not generated from SFRL. In that case the user must be wary of applying the correct level of granularity when specifying the use cases. In practice this is not always the case. However, in order for our mappings to work it is extra important not to blindly define some use cases which seem right, without careful consideration.

Transactions can either be derived from use cases in the Transactions package or from operations in the system interfaces. When we are dealing with manually created models the subfunction level use cases are not likely to be present. In that case the transactions have to be derived from the operations in system interfaces.

When dealing with generated models, based on the stereotypes applied (≪input≫, ≪output≫ and ≪inquiry≫) the kinds of transactions that have to be generated are determined. Based on the include dependencies between these use cases and use cases in the Use Case Model package, we can determine which user goals are supported by which transactions.

When dealing with manually created models there is no way to determine the kind of transaction. In that case we have two options. Either we force the user to manually specify the kind of transformation we are dealing with by not creating transactions of which the kind is unknown, or we choose to assign a certain amount of function points (for example the average of external input and external output) to transactions of which the kind is unknown. A disadvantage of the first option is that this way only indicative counts can be performed. A disadvantage of the second approach is that it is not exactly conform the NESMA standard. We chose to use the second option, because even though it is not exactly conform the NESMA standard, it is still likely to give a more accurate result than the first option, since it enables us to perform estimated counts.

Finally, actors in the Use Case Model should be mapped to users in SFRL. The use cases connected to an actor stand for the user goals that the corresponding user has.

All the mappings as discussed in this section are summarized in Table 3.5. Furthermore, pseudocode for the mapping can be found in Appendix B Section B.5.

UML components model element	***Target model element***	***Name***
Model	System	[Model.name]
Class in Business Concept Model with ≪concept≫ classes in it which are linked to ≪type≫ classes in the Business Type Model	ILF	[Class.name]
Class in Business Concept Model with ≪concept≫ classes in it which are linked to ≪external type≫ classes in the Business Type Model	EIF	[Class.name]
Inner classes of classes in the Business Concept Model with the stereotype ≪concept≫ which are linked to ≪type≫ classes in the Business Type Model	Record in ILF	[Class.name]
Inner classes of classes in the Business Concept Model with the stereotype ≪concept≫ which are linked to ≪external type≫ classes in the Business Type Model	Record in EIF	[Class.name]
≪concept≫ class ≪table≫ ≪type≫ Class in Business Type Model	FPATableILF	[Class.name]
≪table≫ ≪external type≫ Class in Business Type Model	FPATableEIF	[Class.name]
Attribute of ≪table≫ Class in Business Type Model	DET in FPA Table	[Attribute.name]
Attribute of non-table Class in Business Type Model	DET in Record	[Attribute.name]
≪datatype≫ Class in Interface Data Types	DET	[Class.name]
Use Case in in BusinessGoals package	BusinessGoal	[Class.name]
Actor in Use Case Model	User	[Actor.name]
Use case in Use Case Model	UserGoal	[UseCase.name]
Use case in Transactions package with ≪input≫ stereotype	EI	[UseCase.name]
Use case in Transactions package with ≪output≫ stereotype	EO	[UseCase.name]
Use case in Transactions package with ≪inquiry≫ stereotype	EQ	[UseCase.name]

Table 3.5: UML to SFRL mappings summary

3.3.6 Opposite mappings

The mapping of an SFRL model to a UML components model and the mapping of a UML components model to an SFRL model are mappings in the opposite direction of each other. Lets call the mapping of an SFRL model to a UML components model the function f and the mapping of a UML components model to an SFRL model the function g. Ideally, the functions f and g would be exactly each others inverse functions. However, in reality they are not.

If we have an SFRL model m and we apply the functions f and g consecutively to this model we can say that $g(f(m)) = m''$. If f and g would be exactly each others inverse functions then the SFRL models m and m'' are equal to each other. However, we can't say the same for the UML components models m' and m''' when we apply the functions g and f consecutively in the following way: $f(g(m') = m'''$. Even if the functions f and g would be each others perfect inverse it is still possible for m''' to be different from m'. The reason for this is that a UML components model may be elaborated to such an extent that it contains information that can not be mapped to SFRL. That information will be lost when performing $g(m')$. When executing function f on the result of $g(m')$ the lost information can not be recovered and will be missing in the resulting model m'''. What we can say is that $m''' \subseteq m'$.

The functions f and g are probably not exactly each others inverse functions. What we would like to know is the difference between the models m and m'', if any, and the reason for these differences. Also, we would like to know the difference between the models m' and m''' and the reason for these differences.

First we consider the execution of the functions $g(f(m) = m''$. If we look at Table 3.4 and Table 3.5 we can see that all SFRL elements are given a place in the UML components model. Many elements are even mapped to multiple UML components model elements. All the information put into the UML components model by the transformation can be retrieved and if we create an SFRL model based upon this UML components model we get the same SFRL model that we started with.

If we consider the execution of the functions $f(g(m'))$, however, things get a little more complicated. The outcome of the comparison of a UML components model m' and the UML components model m''' resulting from applying the functions is very much depending on the input model m'.

The way in which artifacts are modeled in m' plays a role in the comparison. If the additional stereotypes as we defined in Section 3.3.4 are maintained, then a distinction can be made between user goals and transactions and even the type of transaction can be determined. Also, we will be able to discern files from FPA tables and recognize records in the files. Furthermore we will be able to tell whether the data functions are internal functions or whether they are maintained by an external system. If the additional stereotypes are not maintained, then we will not be able to recognize as many elements. Therefore it is important to maintain the extra stereotypes.

The completeness of the UML components model plays a role. If for example the use case model is defined in the UML components model, but no system interfaces were defined

we get a model in return with use cases as well as system interfaces. However, if the UML components model is fully elaborated into detail, there will be much more information present than SFRL is designed for. After all SFRL was not designed to store every detail of a fully elaborated system. The initial purposes of SFRL were to store the information about a system during a function point analysis in such a way that we can use it as a stepping stone for an initial UML components model and to be able to extract information form a UML components model in order to perform a function point count. The purpose of the mapping from UML to SFRL is to be able to perform a function point count. Further elaboration of the system should be done in the UML components model. If we extract information from a fully elaborated UML components model, then the resulting SFRL model would contain enough information to perform a function point count and to create an initial UML components model based on this information. So in that case m' would be a fully elaborated model and m''' would be an initial UML components model.

Another difference between m' and m''' is the fact that in the business concept model in m' associations between the concepts exist and in the business type model associations between the types exist. These associations can not be determined based on information in SFRL. So when creating a UML components model from SFRL a business concept model and a business type model are created without links between the concepts and links between the types.

What we can say in conclusion to this section is that the functions f and g are not each others inverse functions. If we apply the functions to a model m in the order $g(f(m))$, then the outcome will be the same as the input. However, when we apply the functions on a model m' in the order $f(g(m'))$, the outcome is likely to differ from the input if the extra stereotypes are not maintained in the UML components model and also if the UML components model contains more detail than SFRL can handle or if more artifacts can be derived from the information stored in the UML components model.

3.4 Implementation of the mappings

The mappings as we described in Section 3.3 have to be implemented in order to be able to automatically apply these mappings on an input model and so creating an output model. During this research ATL (ATLAS Transformation language) [2, 7] was used as the transformation language for implementing the mappings in the form of ATL transformations. ATL runs as a plug in on the eclipse platform.

ATL is often called a hybrid model transformation language, in which 'hybrid' refers to the fact that ATL supports both imperative and declarative constructs, the latter of which is preferable whenever possible. The mappings as defined in the previous section can be established by means of transformation rules in which source model elements are checked against the conditions specified in the rules. For each model element matching the conditions of a rule a set of actions, also specified in the rule, are carried out leading to the creation of target model elements. Furthermore, helpers are defined to support the transformation

rules.

For example, consider a situation in which we want to transform all Actors in UML that reside in a package 'Use Case Model' into Users in SFRL. To establish that transformation, we create a rule which selects all Actor elements. According to the UML standard a Package is some kind of Namespace and the ModelElements inside the Package can refer to it by means of an attribute namespace. However namespace does not always have to be a Package, so we have to check that too. In ATL the transformation rule look something like this:

```
rule actor2user {
    from
        a : UML!Actor (
            a.namespace.name='Use Case Model' and
            a.namespace.oclIsKindOf(UML!Package)
        )
    to
        u : SFRL!User (
            name<-a.name
        )
}
```

UML!Actor stands for the model element Actor from the metamodel UML. And the line name<-a.name means that the attribute name of the newly created user is set to the name of Actor a. This is just a very simple example of a transformation rule, but the method for creating more complex transformation rules remains the same. By examining how model elements are linked together in the metamodel, we define how the source of certain information is found in the source model and make sure that it is placed in the correct spot in the target model. This is done for all the mappings as defined in the previous section.

3.5 Input for the model transformations

Once the mappings are implemented as model transformations in ATL, obviously we want to put them to use and see what the results are when we transform some input model into an output model. The transformations for the function point counts and the transformation for mapping SFRL to UML components require an SFRL model as input. The transformation for mapping a UML components model to SFRL, obviously requires a UML components model as input.

The models that we feed to the transformations, the models coming out of the transformations and their metamodels need to be stored in some format. On the platform we used, the MOF 1.4[14] metametamodel from the Object Management Group (OMG) and the Ecore[3] metametamodel from the Eclipse Modeling Framework (EMF) are supported.

The UML 1.4 metamodel can be downloaded from the OMG's website. The function point count metamodel and the SFRL metamodel, however, had to be created. For defining

metamodels we used KM3[1] (Kernel MetaMetaModel), which provides a simple, textual syntax for this purpose. The KM3 models are the serialized in XMI. For the result of the transformation it makes no difference whether MOF 1.4 or Ecore is used. We tried both and they both work fine.

The output models are, of course, created by the model transformations. The models used for input, however, don't just arise out of the blue. We have to create them. For creating UML components models we can use any UML tool that adheres to the UML metamodel from the OMG website and is able to export the UML model in XMI. We used MagicDraw UML Community Edition 9.5.

For creating SFRL models, obviously no language specific tools exist. Initially, we would write SFRL models for testing purposes manually in XMI. However, when the models get larger it is very impractical to do so. If we would create an SFRL model of the information in the hotel reservation system and this model would be stored for example in XMI according to the Ecore metamodel, it might look something like the following example (only one of each type of element is shown in the example)":

In this section we will provide an alternative to the textual creation of SFRL models in XMI which we used initially.

Listing 3.1: Example of SFRL exported to XMI

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
 xmlns:xmi="http://www.omg.org/XMI" xmlns="SFRL">
  <System name="Hotel Reservation System">
    <function xsi:type="ILF" name="Room registration">
      <record name="Room registration">
        <dataElement name="registration no."
                inputTransaction="/0/@function.36"/>
        <dataElement name="room type"
                inputTransaction="/0/@function.41
                        /0/@function.25 /0/@function.32
                        /0/@function.42 /0/@function.36"/>

                    .
                    . (etc.)
                    .

      </record>
    </function>

    .
    . (etc.)
    .
```

```
    <function xsi:type="FPATableEIF" name="Country">
      <dataElement name="country name"/>
      <dataElement name="country code"/>
    </function>

    .
    . (etc.)
    .

    <function xsi:type="BusinessGoal"
        name="Manage reservations"
        supportingUserGoal="/0/@function.12
                /0/@function.11 /0/@function.13"/>

    .
    . (etc.)
    .

    <function xsi:type="UserGoal"
        name="Cancel reservation"
        supportedBusinessGoal="/0/@function.9"
        supportingTransaction="/0/@function.27
                /0/@function.35"
        servedUser="/2"/>

    .
    . (etc.)
    .

    <function xsi:type="EI" name="Alter room info"
        supportedUserGoal="/0/@function.15"
        input="/0/@function.2/@record.1/@dataElement.0
                /0/@function.2/@record.1/@dataElement.1"/>

    .
    . (etc.)
    .

    <function xsi:type="IndependentDataElement"
        inputTransaction="/0/@function.23"
        name="SystemTimeStamp"/>
  </System>
  <User name="Employee" userGoal="/0/@function.12
```

```
        /0/@function.11 /0/@function.13 /0/@function.17
        /0/@function.16 /0/@function.14"/>
</xmi:XMI>
```

It is very undesirable for analysts, or probably for anyone else for that matter, to input the model manually into this textual format. When storing the model in an XMI file according to the MOF 1.4 metamodel things don't look much better. It would be desirable if there was some easier way for inputting the information into an SFRL model by means of a user-friendly interface, which would then export the model into a file that adheres to the MOF 1.4 or Ecore metamodel for example. This can be compared to the variety of UML tools that exist which can be used to create UML models graphically and then export these into textual files.

Possible solutions One idea is to see how analysts currently are used to storing information gathered by them and see if their current method can be adapted in such a way that it can be connected to SFRL. If we consider the situation at Capgemini BAS, the function point analysts use a template spreadsheet to record the gathered knowledge. A connection could be made between this template and SFRL. This way the analysts can just proceed their work in the same way and SFRL models can be extracted from the spreadsheets. However we do not recommend this approach. In the next chapter we will explain why not.

Another idea is to create a tool specifically designed to input the information gathered by function point analysts through a, possibly visual, user interface. In this tool the elements that occur in SFRL might be represented by some visual element on screen (for example each file is represented by an icon). Or buttons for adding, altering or removing information in lists could be used (for example a list of files). To find out how to create the most effective user interface for this purpose is not within the scope of this research. However, for our convenience we came up with a provisional solution for creating SFRL models.

To demonstrate the possibility of using a visual interface we will give an impression of our solution. We borrowed the interface for UML to create our SFRL models. This way we could simply use the already existing UML tools for drawing our SFRL models. We came up with a set of conventions to which the SFRL models in UML should adhere. These conventions describe how the elements from SFRL should be expressed using UML model elements (i.e. which UML model elements should be used for expressing SFRL model elements and which stereotypes should be applied) and the package hierarchy that should be used and a set of stereotypes.

When setting up the package hierarchy we took the SFRL metamodel as a basis. In UML all elements are contained by the root model element 'Model'. An SFRL model is a model about the functionality in a system and the users that use the functionality in the system. As one of the conventions we use the attribute 'name' of the model element 'Model' in UML to record the name of the system. In this Model a package named 'Functions' and a package named 'Users' are created. In the Functions package a package named 'Data Functions' and a package named 'Transactional Functions' are created. In the Data Functions package the packages 'Files', 'FPA Tables' and 'DETs' should be created. In the Transac-

tional Functions package the packages 'Business Goals', 'User Goals' and 'Transactions' have to be created. (see Figure 3.16).

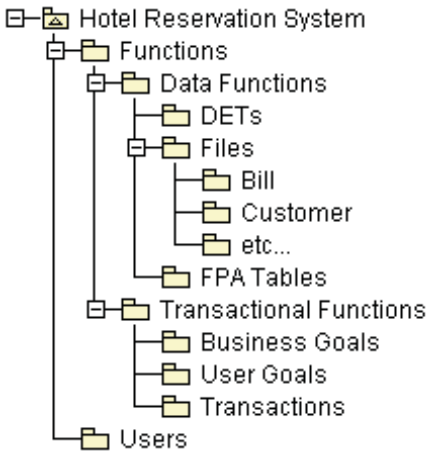

Figure 3.16: Package hierarchy in the UML Interface for SFRL

All files are stored in the package 'Files'. Each file itself is represented by a package. Either the stereotype ≪ILF≫ or ≪EIF≫ should be applied to indicate the type of file we are dealing with. In each file package classes should be created which represent the record types in the files. The stereotype ≪RET≫ should be applied to each record type class. Data elements in the record types are stored as attributes in record type classes.

FPA tables can be created in the package 'FPA Tables', in which each FPA table is represented by a class. Either the stereptype ≪FPA_ILF≫ or ≪FPA_EIF≫ should be applied to indicate the type of FPA table we are dealing with. In each FPA table class attributes can be added which represent the data element types in the FPA tables.

Data element types which are not part of any file or FPA table should be created as classes in the package 'DETs' with the stereotype ≪DET≫.

Business goals are represented as classes in the package 'Business Goals'. The stereotype ≪BusinessGoal≫ should be applied to these classes. User goals are represented as classes in the package 'User Goals'. To these classes the stereotype ≪UserGoal≫ should be applied. The transactions can be created in the package 'Transactions'. These are also represented by classes to which one of the stereotypes ≪EI≫, ≪EO≫ or ≪EQ≫ should be applied. By means of dependencies it is possible to specify which business goals are supported by which user goals and which user goals are supported by which transactions (see Figure 3.17).

Also, by means of dependencies it is possible to specify which data elements are used by which transaction as input and which ones are used as output. 'Business Goals', 'User Goals' and 'Transactions' should be created.

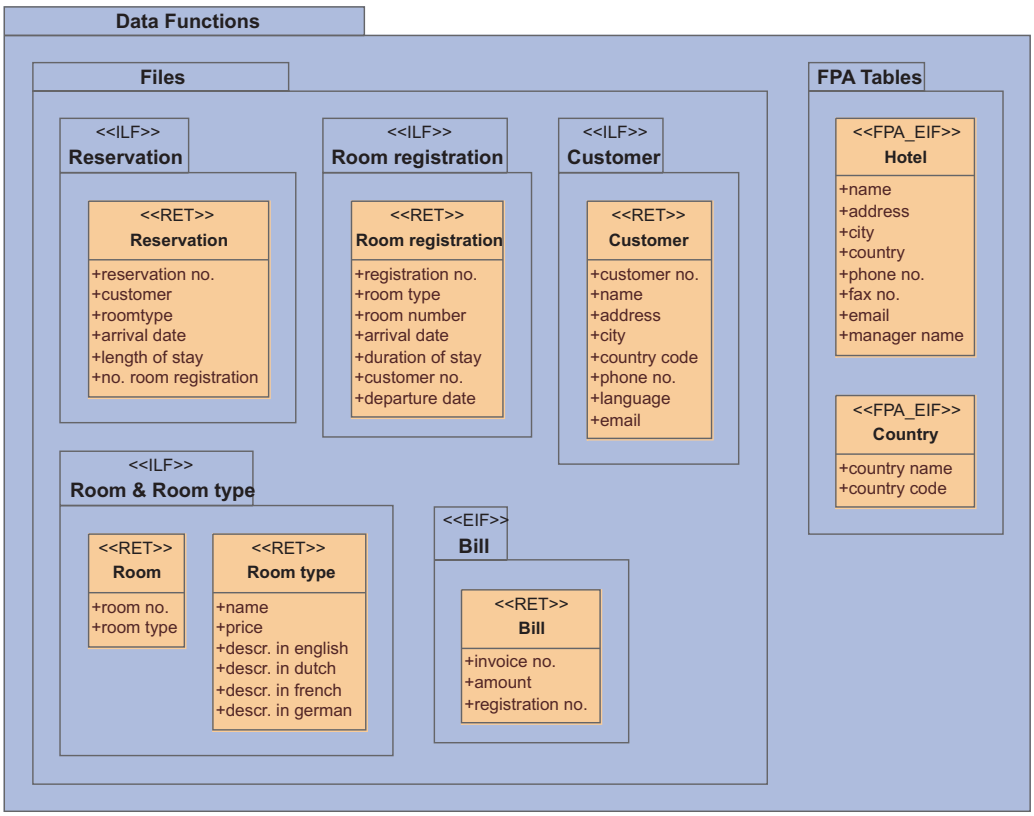

Figure 3.17: Files, records and data elements modeled in UML

Furthermore, we have the packages Business goals, User Goals and Transactions to store the goals at the three levels as defined by Cockburn. Each goal is represented by a class in the appropriate package. The business goals are given the stereotype ≪BusinessGoal≫, the user goals are given the stereotype ≪UserGoal≫ and the transactions are given the stereotype ≪input≫, ≪output≫ or ≪inquiry≫ depending on the type of transaction. The relationships between business goals and user goals and the relationships between user goals and transaction are represented by dependencies (see Figure 3.18).

For each transaction the referenced data elements can be defined by means of dependencies (see Figure 3.19). This way the number of referenced files can be derived for example for determining the complexity of the transaction.

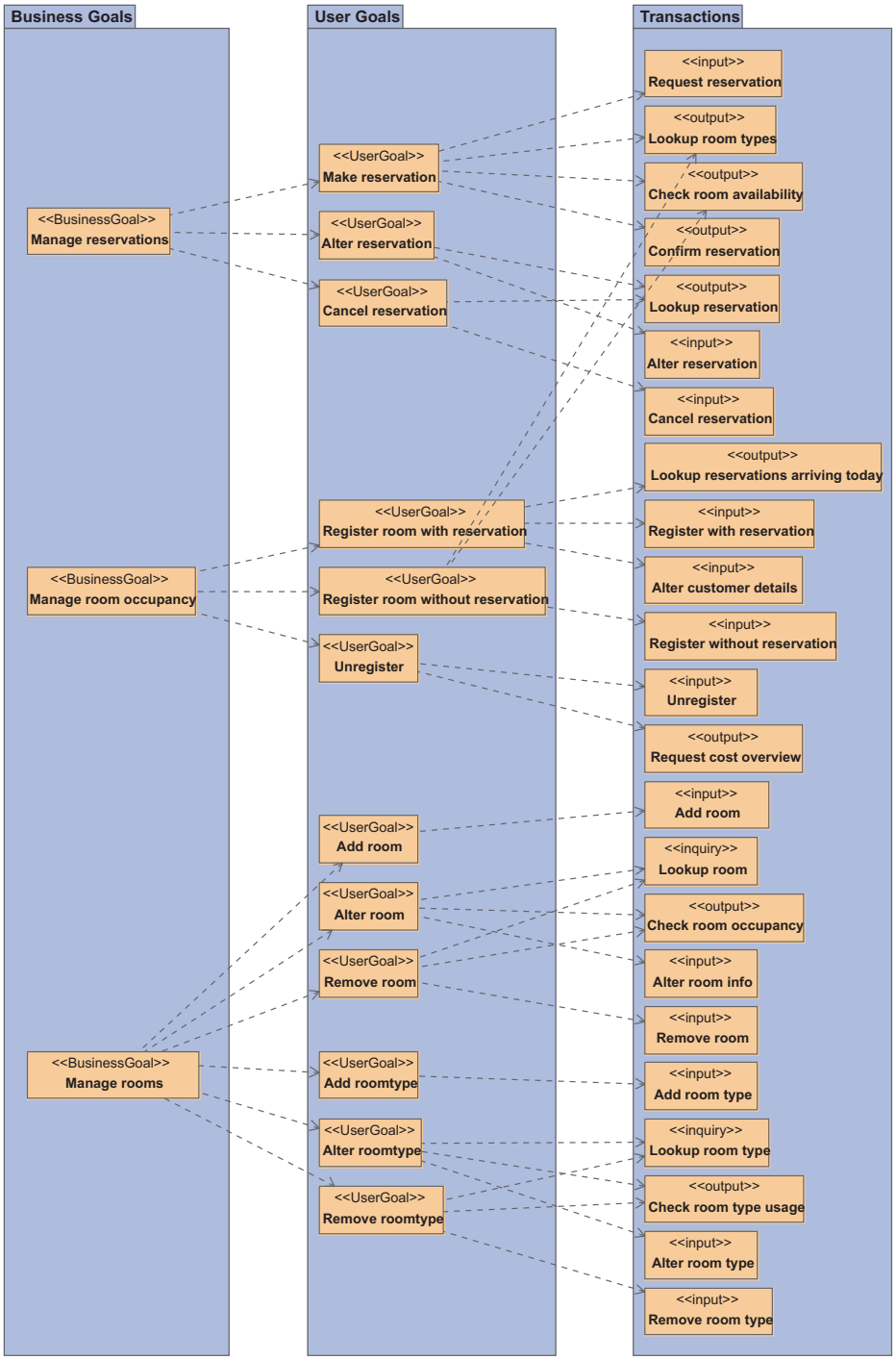

Figure 3.18: Goals at different levels modeled in UML

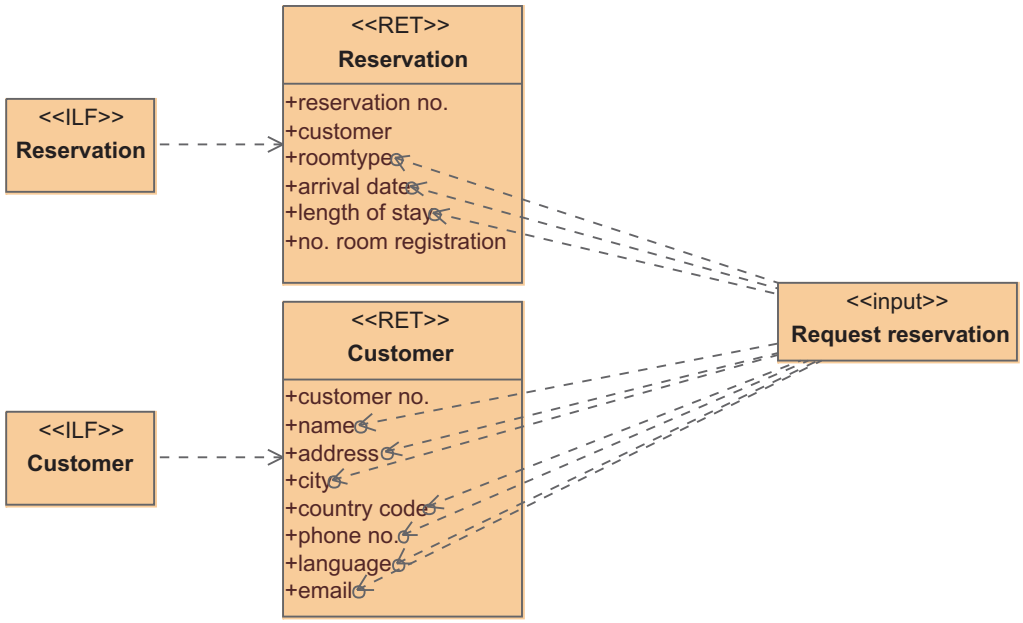

Figure 3.19: Example of a transaction and its connections to data elements modeled in UML

Finally, the relationship between users and user goals should be specified. This is also done by means of dependencies (see Figure 3.20). Now we have covered all elements and relationships that occur in the SFRL metamodel. Each is represented by a graphical model element in UML.

When using UML tools to create our SFRL models, the models are created according to the UML metamodel, because these tools are obviously not designed to store the result as proper SFRL models (i.e. models that adhere to the SFRL metamodel). In order to store the created models as proper SFRL models we created a transformation that converts the ”SFRL in UML” models to proper SFRL models. This solution should give us an idea of what a graphical interface for SFRL might look like. As one can see a good user interface makes the creation of SFRL models considerably easier than creating the models textually in xmi.

At this point we have created a metamodel for SFRL and a metamodel for expressing a function point count, we have provided mappings for performing three types of function point counts based on information in SFRL, we have provided a mapping for generating a UML components model based on information in SFRL, we provided a mapping for generating SFRL based on a UML components model and in this section we provided an easy way for creating SFRL models.

In the next section we will explain how information in SFRL about a system can be used

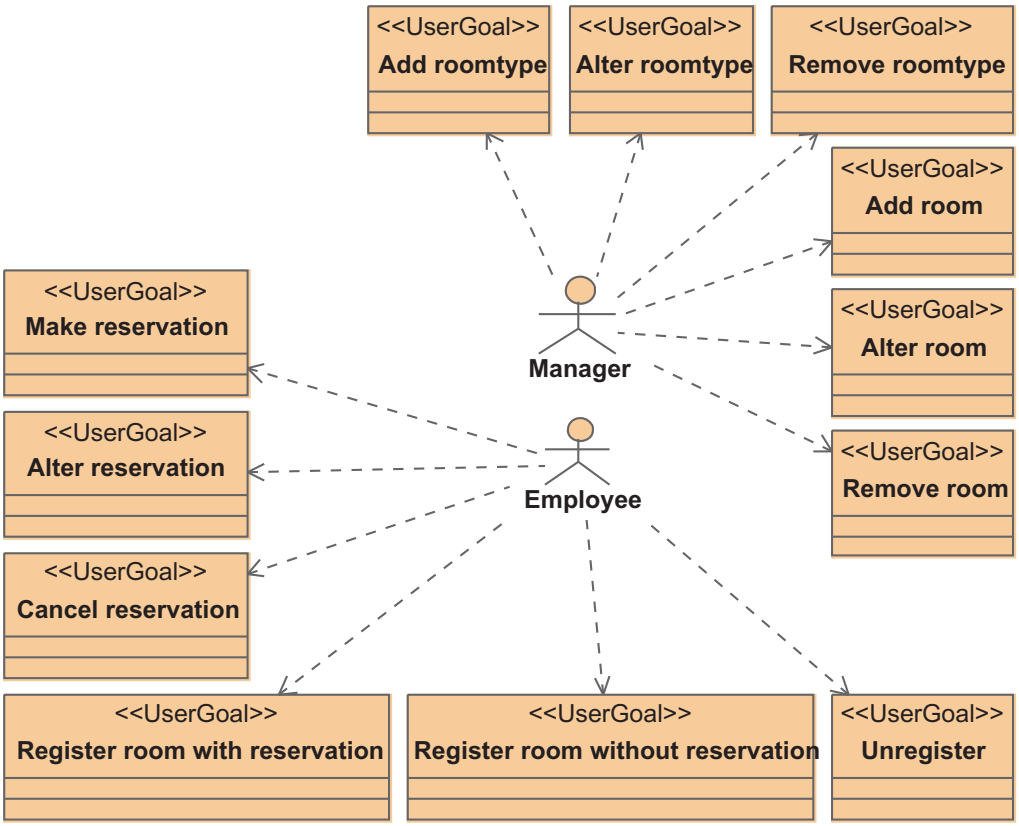

Figure 3.20: Users and their goals modeled in UML

for automatically determining gaps in the specification of the functional requirements.

3.6 Functional Requirements Completeness Evaluation

Function point analysts capture the functionality of a system in order to estimate the size of that system. When performing such an analysis the analysts depend on the information about the system passed on to them. Methods exist that can help in determining the completeness of the requirements.

For instance a CRUD (Create/Read/Update/Delete) matrix [9] can be helpful in locating the gaps in the requirements. A CRUD matrix provides an overview of the data model and the operations performed on the data by the functions in a system. This way it becomes clear which functionality is missing (i.e. it becomes clear whether functionality exists to create, read, update or delete some type of data) or whether some type of data is superfluous (i.e. all desired functionality is available in the system, yet some type of data is not used by the functions present).

When the functionality of a system is captured in SFRL, we can use that information to automatically generate a similar matrix for the files and the transactions. We can specify the expected functionality for each type of file. Each transaction is of a certain type: input, output or inquiry. If it should be possible to create, update and delete information in a certain file, we need at least three input transactions that create input to that file. When we want to read information at least one output or one inquiry transaction is required.

We should keep in mind that these are the minimum requirements. When performing an indicative count, the NESMA [11] supposes that each internal file will have three external inputs, two external outputs and one external inquiry on average. Each external file will have one external output and one external inquiry on average. When sufficient information about the system is captured in SFRL, we can generate a matrix of the relationship between files and transactions, which we can then compare to these expected values.

The hotel reservation contains 4 internal files, 1 external file and several transactions. The completeness evaluation can be easily performed by submitting the SFRL model into a transformation which counts the references from the transactions to the files. These references are actually derived from the references to data elements types, which belong to record types, which in their turn belong to files.

If we generate a matrix from the information in the Hotel reservation system example, this is what we get:

	reserv.	***cust.***	***room & r.type***	***r.registr.***	***bill***
Request reservation	I	I			
Lookup room types			O		
Check room available	O			O	
Confirm reservation	O	O	O		
Lookup reservation	O	O			
Alter reservation	I	I			
Cancel reservation	I	I			
Lookup res. arr. today	O	O			
Register with res.				I	
Alter customer details		I			
Register without res.		I		I	
Unregister				I	
Request cost overview		O		O	O
Add room			I		
Lookup room			Q		
Check room occupancy				O	
Alter room			I		
Remove room			I		
Add room type			I		
Lookup room type			Q		
Check room type usage	O			O	
Alter room type			I		
Remove room type			I		
Total	3I/5O	5I/4O	6I/2O/2Q	3I/4O	1O
Minimum expected	3I/1O/Q	3I/1O/Q	3I/1O/Q	3I/1O/Q	1O/Q
Average expectation	3I/2O/1Q	3I/2O/1Q	3I/2O/1Q	3I/2O/1Q	1O/1Q

Table 3.6: Matrix of the transactions and the data
(I stands for Input, O stands for Output and Q stands for Inquiry)

As we can see, it seems like the specification of the requirements is complete. Yet, the indicative count resulted in 170 points while the estimated and detailed count resulted in an amount which is over 30 points lower. This can be explained by the fact that in this system single transactions are used for input into multiple files or output from multiple files. That way all files are used as expected with less transactions. The slight difference between the detailed and estimated function point count can be explained by the fact that the transactions are less complex on average than expected.

If a metamodel for the matrix is created, the information in SFRL can be mapped onto

a matrix, which can than be used for determining the completeness of the specification of a system.

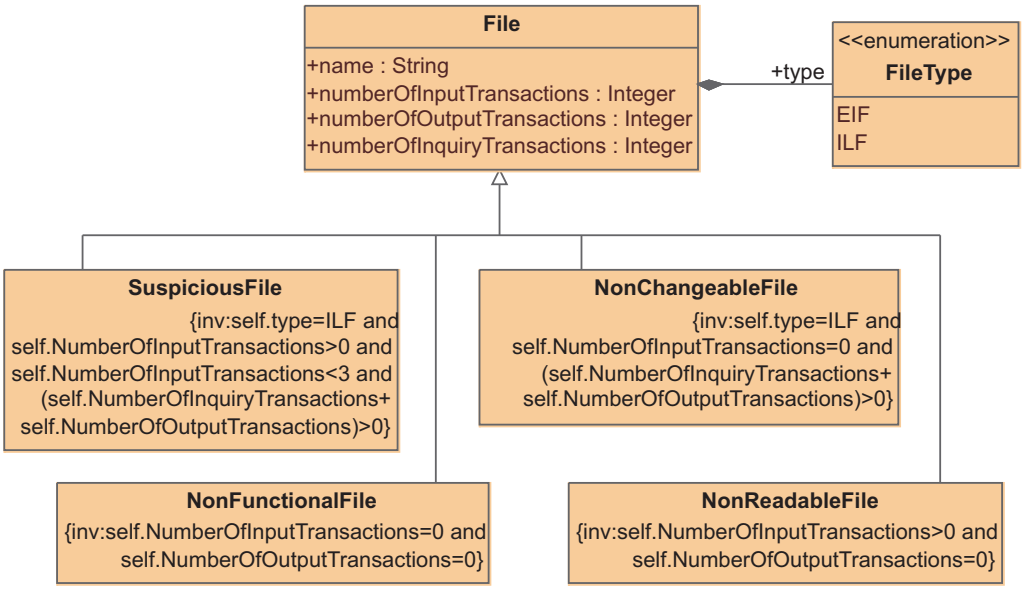

Figure 3.21: Metamodel for recording use of a file

In this model we record the number of references from each kind of transaction to the file. Depending on the kind of file we are talking about, we can distinguish the files that are insufficiently used by the system. As we mentioned above we can look at either the minimum amount of each type of transaction that should use a certain file or we can look at the average according to the NESMA standard. When creating a transformation for detecting a gap in the requirements we have to choose which of these two we want to use as a threshold. The NESMA values are expectations based on averages values, which means that in practice values higher that average may occur as well as values lower than average. So lower values do not necessarily mean that we are dealing with a gap in the specification. Therefore, in our transformation we chose to use the minimum values as a threshold not to mark files as suspicious.

The minimum values are based on the assumption that it must be possible to add, remove and edit information in internally managed files and to read information from all files. If a file is marked as suspicious it does not mean that definitely some functionality is missing for managing that file. It is possible to decide that no functionality is required to edit information in a certain file, but only functionality to add and remove information.

We made a distinction between various kinds of suspicious files:

- NonFunctionalFile: file which is not used at all,
- NonReadableFile: file that can be changed but which can not be read,
- NonChangeableFile: ILF which can be read but can not be managed,
- SuspiciousFile: ILF which can be read but does not seem to have enough management functionality.

Using this method we get a clear overview of the files that are either not necessary for the system or for which transactions have to be defined in order to complete the functional requirements of the system.

Summary

In this section, we first determined what kind of transformations we needed to create. The next step was to come up with the metamodels required to create the transformations. By creating a metamodel for SFRL we specified what SFRL should look like. We described a hotel reservation example system which was used for exploratory investigation. Also, we described the mappings we needed to create.

Furthermore, we came up with an interface for creating SFRL models. This interface consists of the UML user interface in which the SFRL model can be drawn according to a set of rules and a transformation which transforms the created UML elements into an SFRL model.

Finally we came up with a solution for using the SFRL model for evaluating the completeness of the functional requirements. This solution consists of a transformation which transforms the SFRL model into a matrix in which the presence of various types of transactions per file is verified.

In the next chapter we will evaluate the transformations created in this chapter.

Chapter 4

Evaluation

In the previous chapter we elaborated SFRL and some transformations for function point counting, creation of a UML components model based on information in an SFRL model, creation of an SFRL model based on a UML components model and a functional completeness evaluation.

In this chapter we evaluate the usefulness and usability of SFRL and the transformations using example systems.

4.1 Success criteria

When evaluating the use of SFRL and the transformations created during this project, success criteria must be defined. Also, we need to specify how we found out to what extent these success criteria are met.

One of the main ideas of this project is that a formalized way of keeping the knowledge about a system gathered during a function point analysis enables us to use that knowledge in another stage of a software project without having to analyze the system all over again, which is fairly time consuming. This formalized way of keeping knowledge, implemented in the form of SFRL, could save us precious time provided that the creation of SFRL models of systems itself is not too time consuming. From the company's point of view it is interesting to know whether the current method for recording function point analyses, namely a spreadsheet template, can play a role in the creation of SFRL models. In Chapter 3 we described the user interface that we used for creating SFRL models. We also want to know how the use of this interface turned out. In Section 4.3 we evaluate this subject.

If at some point in a software project we want to recount the function points in the system we can create an automated recount instead of manually performing a time consuming function point analysis. In order to do so the UML components model is mapped to an SFRL model. From this model function point counts can be generated. What we want to know is whether the automated recount is comparable to the initial function count of a system. If the UML components model of the system is not changed, we want the regenerated SFRL model to be the same as the initial SFRL model from which the UML components

model was generated. If the initial function point counts and the function point recounts are consistent, changes in the amount of function points can only be caused by changes to the system. To test this we compare the regenerated SFRL model to the initial SFRL model (see Figure 4.1). This way we can discover possible differences between an initial function point count and an automated recount of the function points, although we do expect these to be the same.

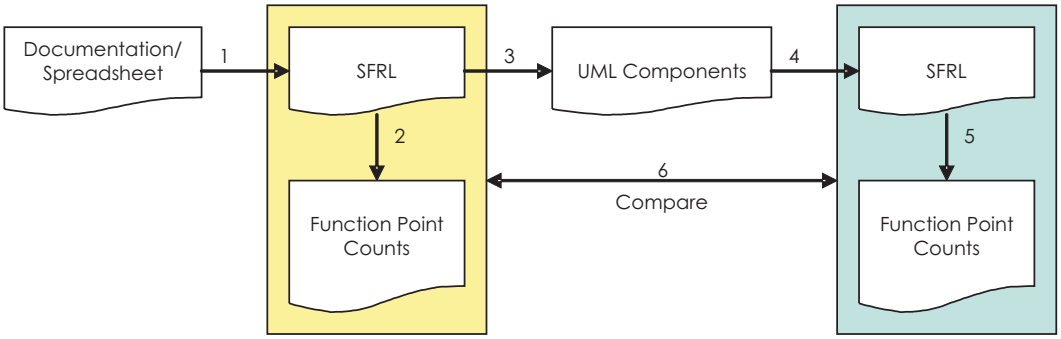

Figure 4.1: Comparing an initial SFRL model to a regenerated SFRL model

The transformation from a UML components model to an SFRL model differentiates between UML components models that were generated from SFRL and traditional UML components models by looking for the presence of the extra stereotypes we defined in Chapter 3 (see Section 3.3.4). If we have a manually created UML components model from an existing software project, we might want to perform a function point count of that system with the UML components model as a starting point. We want to test the transformation from such a basic manually created UML components model to SFRL and see what kinds of function point counts can be performed with such a UML components model.

When a UML components model is transformed to SFRL it is possible to subsequently transform the generated SFRL model back to a UML components model. SFRL is not designed for this purpose, but a reason why we could want to do this is when we have a simple UML components model as a starting point like we just described and turn it into a UML components model which is modeled according to our slightly modified UML components profile. It is interesting to compare the UML components models and see what changes even if we do not alter the SFRL model of the system (See Figure 4.2).

Furthermore, we created a transformation which performs a simple functional completeness evaluation on an SFRL model. We want to know whether the function completeness evaluation generates a useful result when we use it on our test systems.

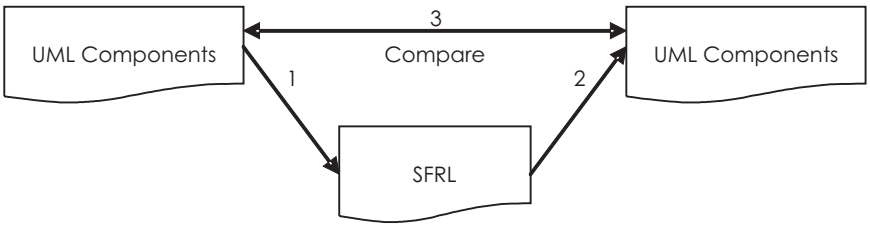

Figure 4.2: Comparing an initial UML components model to a regenerated UML components model

4.2 Cases

For evaluating the transformations we use the hotel reservation system from the previous section and two practice cases for which estimated function point counts are available. The hotel case has already been described in detail in this thesis. In the previous section we listed the files and transactions in the system. The transactions are grouped by user goal and the user goals are grouped by business goal. The complexity of files and transactions and the relationships between the files and transactions are described in Appendix A. The other two sample systems are not described in detail in this report.

The purpose of the first system (called System 1 in the sequel) is to register the use of hardware. For this system only an indicative count was available in the form of a spreadsheet along with documentation which contains use case descriptions. Based on this documentation we created an estimated count and had it verified by a function point analyst. The subject of the second system (called System 2 in the sequel) is confidential. For this system we were only provided with a spreadsheet in which an estimated count was elaborated.

A full listing of several pages of all the functions in the systems does not add any value to this report. We will summarize the information that matters for our evaluation, such as the number of external inputs present before performing the transformations and the number of external inputs after running the transformations.

4.3 Capturing the systems in SFRL

Before we can run the tests we have to capture the test cases in SFRL. To do so we used the UML User Interface described in Section 3.5. We will compare this to the current way function point analysts archive their knowledge about a system.

For the hotel system we created the SFRL model by drawing the diagram that can be seen in Figure 3.17 for modeling the files in the system and the diagram in Figure 3.18 to model the transactional functions in the system. To model the links between transactions and files a diagram like the one in Figure 3.19 had to be drawn for each transaction and to model the users and their relationships with the transactions the diagram in Figure 3.20 had to be drawn. These diagrams are created by consulting Tables 3.1 and 3.2 and the

information in Appendix A. These diagrams are the transformed to actual SFRL models.

For System 1 a spreadsheet was available in which a list of files and a list of user goals were present. The transactions required for the user goals could be derived from the use case descriptions in the documentation of the system. Since no detailed count was available no diagrams for the relationship between data functions and transactions could be made. Also no information about the data elements is available.

For System 2 a spreadsheet was available with a list of files and a list of transactional functions. This was sufficient to create diagrams for the files and transactional functions, but also in this case no information about the data elements in files and the relationships between the files and transactions was available.

We encountered no problems while capturing the systems in SFRL. All information about the system that we needed to capture to perform function point counts could be captured.

Being the one who created this UI, for the author it was easy to use. In this UI elements are connected by arrows. This is easy for entering a new system, but when a system has a lot of connections the result looks like a mess of arrows and it may become hard to get a quick overview of what is going on in the system and to make changes to the system. For a function point analyst, one can imagine that this interface does not have a familiar look and feel. Furthermore, for this UI we used UML, which is a general purpose modeling language. Because of the nature of UML it is hard to implement the UI in a totally fool proof way which is convenient for a function point analyst. For testing and demonstration purpose this UI was good enough, but for use in practice a better manageable user interface is recommended.

At Capgemini BAS function point analysts currently record the results of their function point analysis in a spreadsheet. The main parts of this spreadsheet are a list with data functions and a list with transactional functions. The complexity of data functions can be specified by entering the number of records and the number of data elements in a file. The complexity of transactional functions can be specified by entering the number of referenced data elements and the number of files referenced. This is a very simple way of recording the results of a function point count, but it is sufficient to determine the amount of function points in the system for all three types of function point counts and it provides a nice and clean overview of the functionality in the system. A disadvantage of this simple way of recording information is that a lot of gathered information gets lost, especially when performing a detailed function point count. For example, for determining the complexity of a file the records in a file and the data elements in each record are identified and counted. This knowledge of each individual record and data element is thrown away and only the amounts are recorded. In SFRL on the other hand we aimed to preserve this kind of information.

In the previous section we mentioned the possibility of extracting information from the spreadsheet into an SFRL model. This, however, comes with some problems.

- In the spreadsheet there is one sheet for listing the transactional functions in the system, but there is no formal way in which user goals and transactions are recorded. Function point analysts are free to record user goals the way they like. The same

holds for files and FPA tables which are all listed on one sheet. The fact that no formal way is defined for this purpose is not convenient for automatically extracting information from the spreadsheet.

- Another problem is the fact that the spreadsheet does not support users and links between the users and user goals. This means that this feature of SFRL remains unused.

- A larger problem is the fact that per file only the number of records and the total number of dets in the file are recorded in the spreadsheet. There is no support for actually mentioning the records in the files and the data elements in each record, while in SFRL the actual records and the data elements in each record are recorded.

- Furthermore, there is no support for the links between transactions and data elements. Here also only the number of data elements and the number of file types referenced are recorded. The spreadsheet does not offer a way to record which data elements are referenced by which transaction, not to mention whether these data elements are used as input or output by the transactions.

As we can see there are some problems with using the spreadsheet as a basis for creating SFRL models. We can overcome the first problem simply by specifying way in which function point analysts must distinguish user goals and transactions in the spreadsheet.

The second problem could be solved by introducing a new sheet for listing the users and the user goals that are linked to these users.

A simple workaround for third problem could be to create a dummy record in SFRL for each record counted in the spreadsheet. For example for a file named room with two records the records room_record1 and room_record2 can be created. But then we still would not know how many data elements are in each record. We only know the total amount of data elements in the file, but it is unknown to which record each data element belongs.

If we would randomly distribute dummy data elements among the records, when we generate a UML components models the generated models don't make any sense. We would for example get a business type model in which the types have the names of dummy records and the attributes in the types would have the names of dummy data elements of which we would not even be sure whether these are supposed to be in that record or not. This does not seem like a good solution.

Another possibility is to actually specify a way to list the records and the data elements in each record in the spreadsheet. Then, the information in the sheet becomes more useful, but a disadvantage is that this would make the overview of data functions less orderly.

The last problem is something that is hard to solve in the spreadsheet in a manageable way. Every transaction can maintain references to multiple data elements and every data element can be referred to by multiple transactions. It is hard to capture these references without making a mess of the spreadsheet.

All in all, we see that extracting information from the spreadsheet and generating an SFRL model with that information may come with some difficulties. When we look at the kind of information that is involved with these problems, we see that most of these problems

come up when we are dealing with a detailed function point count. With an indicative count we just have to deal with a list of data functions and with an estimated count a list of transactional functions comes on top of that. This is information that can very well be stored in the spreadsheet. For creating an SFRL model of information from indicative counts and estimated counts the spreadsheet suffices. Since by far the most function point counts are either indicative or estimated function point counts in most cases we could use the spreadsheet for automated extraction of the information to SFRL, provided that the way information is entered in the spreadsheet is formalized. For creating an SFRL model of information from a detailed count we do not recommend using the spreadsheet as input. To be able to specify information from a detailed count in SFRL a tool with a user interface especially designed for creating SFRL models is recommended.

4.4 Initial function point counts

After recording our test systems in SFRL we performed function point counts using our transformations. First we performed the indicative function point counts for all three systems. An indicative count is based on the number of ILFs and EIFs and whether or not FPA tables are present. In Table 4.1 the results of the indicative counts can be seen.

	System 1	***System 2***	***Hotel System***
Number of ILFs	7	14	4
Number of EIFs	0	2	1
Number of FPA tables ILF	3	2	0
Number of FPA tables EIF	0	2	2
Number of Function Points	280	570	170

Table 4.1: Indicative function point counts

Besides the data functions, an estimated function point count is also based on the amount of external input, external output and external inquiry transactions. In Table 4.2 the results of the estimated function point counts can be seen.

	System 1	***System 2***	***Hotel System***
Number of ILFs	7	14	4
Number of EIFs	0	2	1
Number of FPA tables ILF	3	2	0
Number of FPA tables EIF	0	2	2
Number of EIs	38	113	13

continued on next page

continued from previous page			
Number of EOs	27	44	8
Number of EQs	0	0	2
Number of Function Points	356	805	138

Table 4.2: Estimated function point counts

When performing a detailed function point count the complexity of the functions has to be determined. To be able to do so information about the records and data elements in data functions and information about the data elements that are referenced by the transactions must be available. For the hotel case the SFRL information is detailed enough to perform a detailed function point count besides the indicative and estimated function point counts. For the other two systems only indicative and estimated function point counts could be performed. In Table 4.3 the results of the detailed function point count of the hotel system can be seen.

	Hotel System
Number of ILFs with low complexity	4
Number of ILFs with average complexity	0
Number of ILFs with high complexity	0
Number of EIFs with low complexity	1
Number of EIFs with average complexity	0
Number of EIFs with high complexity	0
Number of FPA tables ILF	0
Number of FPA tables EIF	2
Amount of function points for FPA tables ILF	0
Amount of function points for FPA tables EIF	5
Number of EIs with low complexity	9
Number of EIs with average complexity	4
Number of EIs with high complexity	0
Number of EOs with low complexity	4
Number of EOs with average complexity	3
Number of EOs with high complexity	1
Number of EQs with low complexity	2
Number of EQs with average complexity	0
Number of EQs with high complexity	0
Number of Function Points	125

Table 4.3: Detailed function point count

4.5 Recounting the function points

The set of initial function point counts is the point of reference in this evaluation. In this section we perform some transformations and then recount the function points in the systems. First we transformed the SFRL models of the systems to UML models. Then the UML models were transformed back to SFRL. The results of the recounts are compared to the initial function point counts. This will show us whether the function point recounts and the initial function point counts are consistent.

First of all we started with performing indicative function point recounts. For an indicative function point count only the data functions are required. As can be seen in Table 4.4 the indicative recounts are exactly the same as the initial function point counts. The data functions are restored exactly to the same state they were in, in the initial SFRL models.

	System 1	***System 2***	***Hotel System***
Number of ILFs	7	14	4
Number of EIFs	0	2	1
Number of FPA tables ILF	3	2	0
Number of FPA tables EIF	0	2	2
Number of Function Points	280	570	170

Table 4.4: Indicative function point recounts

Subsequently we tested the estimated function point recounts. In an estimated function point count besides the data functions also the transactions are required. In Table 4.5 the results of the estimated function point recounts can be seen. These results are also exactly the same as the initial estimated function point counts. The transactions are all restored exactly to the state they were in, in the initial SFRL models.

	System 1	***System 2***	***Hotel System***
Number of ILFs	7	14	4
Number of EIFs	0	2	1
Number of FPA tables ILF	3	2	0
Number of FPA tables EIF	0	2	2
Number of EIs	38	113	13
Number of EOs	27	44	8
Number of EQs	0	0	2
Number of Function Points	356	805	138

Table 4.5: Estimated function point recounts

Finally we performed a detailed function point recount of the hotel reservation system. The results of this recount can be seen in Table 4.6. As we can see here this table also shows us the same results as the initial detailed function point count.

	Hotel System
Number of ILFs with low complexity	4
Number of ILFs with average complexity	0
Number of ILFs with high complexity	0
Number of EIFs with low complexity	1
Number of EIFs with average complexity	0
Number of EIFs with high complexity	0
Number of FPA tables ILF	0
Number of FPA tables EIF	2
Amount of function points for FPA tables ILF	0
Amount of function points for FPA tables EIF	5
Number of EIs with low complexity	9
Number of EIs with average complexity	4
Number of EIs with high complexity	0
Number of EOs with low complexity	4
Number of EOs with average complexity	3
Number of EOs with high complexity	1
Number of EQs with low complexity	2
Number of EQs with average complexity	0
Number of EQs with high complexity	0
Number of Function Points	125

Table 4.6: Detailed function point recount

Due to the way everything is mapped into a UML components model, all information of the SFRL model is preserved and can be restored exactly to its original state. We also compared the contents of the initial SFRL model to the SFRL model generated from the UML components model that was generated from the initial SFRL model by viewing both models in the Ecore model editor in Eclipse. The content of the models appears to be identical.

What we can conclude from this is that for these systems, differences between the initial function point counts and function point recounts in a later stage in a software project are not caused by the reverse transformation not being able to restore the SFRL model. Possible differences in the amount of function points must be the result of changes in the UML components models.

4.6 Use of a simple manually created UML components model

Information in UML components models generated from SFRL is presented in a slightly different way than in UML components models created from scratch. For example a difference between a conventional business concept model and a generated business concept model exists due to the fact that the distinction between files, records and FPA tables is preserved in the generated business concept model, while this distinction does not exist in a conventional business concept model. Therefore, in this transformation, a distinction is made between UML components models generated from SFRL and other UML components models.

UML components models generated from SFRL models have already come to attention in the sequence of transformations from SFRL to UML to SFRL. In this section we want to see what a UML components model created from scratch looks like in SFRL and what kind of function point counts can be performed based on these models.

For this test we used the simple manually created UML components model of the hotel reservation system (see Figures 3.11 and 3.12) with only a business concept model, a use case model and a business type model. As a result, an SFRL model was generated with only files and user goals. Each file has only one record with the same name as the file. Traditionally, there are no use cases at transaction-level in the UML components model. Therefore, we can only perform an indicative function point count based on this UML components model. The results of this function point count can be seen in Table 4.7.

	Hotel System
Number of ILFs	5
Number of EIFs	3
Number of FPA tables ILF	0
Number of FPA tables EIF	0
Number of Function Points	220

Table 4.7: Indicative function point recount based on a manually created UML components model

As one can see there are some differences between this indicative function point count and the indicative function point count which had the SFRL model as a starting point. In this case the indicative function point count counted 5 ILFs and 3 EIFs and no FPA tables, while in the indicative count based on the SFRL model only 4 ILFs were counted, 1 EIF and 2 FPA tables EIF.

One thing that we noticed is that in this case no distinction can be made between FPA tables and files. All concepts that should have been counted as FPA tables are now counted as files, which causes a higher amount of function points.

Another thing that can be noticed is that there are no files with multiple records. While in the SFRL model the concepts Room and Room type are represented by records that are

part of the same ILF, in this case they are translated to records in separate ILFs, because originally the UML components method does not provide a way to group records. Because of this an extra ILF is counted which results in a higher amount of function points.

Even though the amount of function points as a result of this count is higher than it is supposed to be it still is better than nothing. The amount of FPA tables in a system usually is not very large and most files in a system have only one record. Indeed the amount of function points is too high, but it still approximates the correct function point count of the system.

4.7 Regenerating a UML components model

Consider the situation in the previous section with a manually created UML components model and an SFRL model which was extracted from this UML components model. When further elaborating the system we can choose to continue working with the manually created UML components model, or we could choose to regenerate a UML components model from the SFRL model, possibly after modifying this SFRL model, and continue working with the newly generated UML components model.

In this section we compare both options. From the SFRL model generated in the previous section we generate a UML components model again and compare this to the manually created UML components model.

Note however that SFRL was not designed with the purpose of regenerating UML components models from SFRL models that have been extracted from UML components models. When dealing with a simple manually created UML components model from an early stage of the software project, the amount of informations that is lost in the transformation to SFRL is limited, but when dealing with an elaborated UML components model, quite a lot of information will get lost in that transformation since SFRL can not contain all the information that is potentially stored in a UML components model. Since we are dealing with a very simple UML components model this is not an issue in this case.

When comparing the original UML components model to the regenerated UML components model we came across some differences between the models. In the original model relations between the concepts in the business concept model are modeled. These relations can not be stored in SFRL and are therefore missing in the regenerated business concept model. So this is a disadvantage of regenerating UML components models.

Another difference is the fact that in the original business concept model all classes are concepts, while in the regenerated business concept model all classes are marked with the stereotype ≪FPA File≫. These classes have an inner class with the same name with the stereotype ≪concept≫. The cause of this difference is that the transformation from UML components to SFRL maps all concepts to files since no distinction can be made between concepts which should be mapped to files and concepts which should be mapped to FPA tables. When mapping the generated SFRL model to a UML components model the

additional stereotypes as listed at the end of Section 3.3.4 are applied in order to preserve the distinction between files and FPA tables. As business concept models the two models are practically equal. What we have to be careful about, however, is that because of the additional stereotypes in the regenerated model it seems like we are unmistakably dealing with files and there has already been thought about which concepts should be grouped as records in files. This is clearly not the case since that information was not present in the first place. Therefore it would be wise to carefully consider the correctness of the grouping of records in files and whether files really should have been files instead of FPA tables and make corrections to the SFRL model or the regenerated UML components model if necessary.

The last difference that we came across was the fact that while in the initial model no system interfaces were present, in the regenerated UML components model system interface types have been defined for each user goal in the system.

4.8 Functional completeness

In the previous chapter we defined a functional completeness evaluation method, which checks whether the functionality in a system seems complete, based on the references from transactions to files. This means that a functional completeness evaluation can only be performed if information from a detailed function point count is available (i.e. if references exist between the files and the transactions). If we perform a functional completeness evaluation on System 1 and System 2, all files would be marked as unused, because they are not referenced by any transaction at all. So when only information from an indicative or from an estimated count is available, a functional completeness evaluation is not useful.

The only system for which we can perform a functional completeness evaluation is the hotel reservation system. In the previous chapter we have already created a table with the references from transactions to files for this system (see Table 3.6). Since all files are sufficiently referenced not to be suspicious, all files are displayed as normal files in the output. When we remove some references from transactions to a file and rerun the functional completeness evaluation, the affected file was marked correctly as either one of the variants that indicates possible missing functionality (i.e the file was not read or could not be altered or a combination of both).

What we can say about the functional completeness evaluation is that it works correctly, but sufficient information about the system has to be available for the evaluation to be useful.

4.9 Usability and usefulness

In this section we will consider the usability and the usefulness of SFRL. What difference does SFRL make for function point analysts? What difference does it make to system specifiers? What could improve the usability of SFRL?

For function point analysts (almost) everything remains the same. They analyze the same kind of documents they always did. The way in which they analyze these documents also remains the same. The information discovered in the documents remains the same. The only thing that may change is the way they record the results of their analysis. But even that does not have to be a great change if we would choose to sacrifice recording information supported by SFRL but not supported by the spreadsheet.

It is thinkable that some new tool is created that stores the entered information directly in SFRL. This tool may have an interface that looks familiar to the template spreadsheet currently used by function point analysts. Another option is to create export functionality that extracts the information from the spreadsheet and stores it in SFRL. This way the way in which function point analysts work remains completely the same. Although in the latter case it would be desirable to agree upon a way on how to support the goal levels in the spreadsheet template. This approach is only usable for recording information from an indicative or an estimated count. the spreadsheet template is too simplistic to support creation of an SFRL model of information from a detailed count.

For the system specifiers things do change. Instead of starting with an empty UML project they begin with a generated UML components model in which a lot of information has already been entered. This model is a starting point for further elaborating the system specification. Also, when further elaborating the UML components model, it is desirable to keep the information recognizable for the model transformations to be able to optimally use the information for function point counts.

The platform we now use to perform the transformations is not the desired way of working. For each project we would have to create new run configurations. Its mere purpose is to experiment and to demonstrate the approach. For actually deploying this approach it is more likely that an application especially designed for creating, saving and loading SFRL models, with built in support for the model transformations is made. If such an application would be made this approach becomes very usable.

Chapter 5

Conclusion

In this thesis we described the issue of knowledge about the functionality of a system not being used optimally in the software production process. Our main research goal was to create a process in which knowledge gained during the function point analysis can be shared so it can be useful for creating a system specification and vice versa. In order to facilitate this we came up with SFRL, a formal language for capturing this knowledge in a model. Once the knowledge is captured in SFRL it can be used for several purposes. In this chapter we will discuss to what extend the research goals (see Section 1.5) are met.

First we had to determine what SFRL should look like. Since SFRL is an intermediate format mainly between a NESMA function point count and a UML components specifications we had to examine these standards in order to find out what needs to be recorded in SFRL.

In Chapter 2 we discussed the NESMA standard for function point analysis and the UML components method for specifying a system and we also discussed the connection between these. Three types of function point counts are available for determining the system size. Each type of count requires a certain level of detail of the available information about the system. On the lowest level only files in a system have to be identified. One level higher also transactions are identified. On the highest level the complexity of the files and transactions is identified. The level of detail of a UML components model depends on the amount of information available about a system.

What became clear was that in both the NESMA standard and the UML components method there is a data side and an actions side to a system which both have to be captured in SFRL. However, the granularity level of the actions side in the NESMA standard did not match the granularity level of the actions side in the UML components method. To resolve this we adopted the goal levels as defined by Cockburn [5] in SFRL.

In Chapter 3 we further discussed from what kind of information SFRL is built up and came up with a metamodel which defines what elements are present in an SFRL model and the relationships between these elements. We defined mappings from SFRL to function point counts, from SFRL to a UML components model and from the UML components model to an SFRL model. Additionally we defined a mapping from SFRL to a functional completeness evaluation. Furthermore, we defined an example user interface for creating

SFRL models.

In chapter 4 we experimented our approach on an example hotel reservation system and two test systems. With our experiments we showed that the efficiency of the analysis of these systems increased: The information about the system which we recorded in SFRL was not only used for function point analysis, but was also put to use for successfully creating an initial UML components specification of the system and for performing a functional completeness evaluation. Furthermore, knowledge sharing was also beneficial for counting the function points based on a UML components specification.

In our experiments we have also shown that the information mapped from SFRL to a UML components model can be fully recovered back to SFRL. This is achieved by applying some additional stereotypes to the UML components model in order to preserve information that would otherwise not have a part in the UML components method. Without these stereotypes, in the business concept model we would not be able to distinguish files from FPA tables, in the use case model we would not be able to tell whether transactions are external input, external output or external inquiries, and in the business type model we would not be able to model external data functions and therefore information about data elements in these external data functions would get lost.

When transforming from a standard UML components model to an SFRL model we will lose information, since the UML components is designed to contain much more information than SFRL. Also, in a standard UML components model without the additional stereotypes we can not differentiate between concepts that should be mapped to files and concepts that should be mapped to FPA tables. In order to approximate a proper function point mapped all concepts to files. This results in a slightly higher function point count than a proper function point count. For a quick indication of the function points this is better than nothing, but what we have to keep in mind is that the resulting value can be even less accurate than an indicative count, which is not accurate itself.

Regarding the consequences for the software production process we can say that basically the function point analysis itself does not change. The only difference for function point analysts is that the result of their efforts has to be recorded in SFRL. If a link is made between the spreadsheet tool and SFRL nothing at all changes for the function point analysts. Otherwise the only difference is that they have to use another tool to record their findings.

The difference for system specifiers is that they don’t start from scratch with the UML components models. Instead, a basic UML components model is generated from the information in SFRL which has to be further elaborated by the system specifiers.

An additional benefit to this approach is that a automated functional completeness evaluation can be performed if enough details are captured in SFRL. This can be done on basis of a matrix similar to a CRUD matrix. On the data side of the matrix the files are put side by side. On the functionality side of the matrix, the transactions are listed. For creating, updating or deleting information in files external input functions are expected, while for reading

information from files an external output or an external inquiry function is expected. This way we can determine the minimum expectations per file. An indicative count is based on average expectable functionality per file. By looking at the way an indicative count is built up according to the NESMA guidelines, we can determine the average amount of transactions per type that can be expected. Based on these expectations we can determine whether files should be marked as suspicious or not. For the suspicious files we can then easily find out whether all functionality for using the data in the file is present.

5.1 Future work

We designed SFRL for recording information about a system gathered during a function point analysis. What we might say is that in this process the documentation about the system is manually transformed into an SFRL model. It would be convenient if this step could be automated. An idea for doing so could be to define a formal language for specifying the requirements of a system. This is something that could be interesting for future research.

As we stated at the beginning of this thesis, SFRL is an intermediate form in which information about a system is stored, which can then be used for multiple purposes. Another interesting subject for future research is the use of SFRL with other functional size measurement methods besides the NESMA standard and for use with other system specification methods besides the UML components method.

Furthermore, we already came up with a way for creating SFRL models. In Section 4.3 we concluded that this way of creating SFRL models is good enough for testing and demonstration purpose, but for use in practice a better manageable user interface would be desirable. Future research on this subject might deliver a convenient user interface for creating and presenting information in SFRL models.

It would also be desirable to support SFRL in terms of tooling. For example in sequel to research to a convenient user interface a tool can be created which can be used to create and manage SFRL models, provide function point counts based on the recorded information, which can automatically evaluate the functional completeness of the specified system and which could automatically create UML components models based on the SFRL models.

Bibliography

[1] LINA & INRIA Nantes ATLAS group. *KM3: Kernel MetaMetaModel*, 2004.

[2] LINA & INRIA Nantes ATLAS Group. *ATL User Manual, version 0.7*, 2006.

[3] Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison-Wesley Professional, 2003. (ISBN 0 131 42542 0).

[4] John Cheesman and John Daniels. *UML components: a simple process for specifying component-based software*. Addison-Wesley, 2000. (ISBN 0 201 70851 5).

[5] Alistair Cockburn. *Writing effective use cases*. Addison-Wesley, 2000. (ISBN 0 201 70225 8).

[6] International Function Point User Group (IFPUG). *IFPUG Function Point Counting Practices Manual — Release 4.2*, 2004.

[7] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *Satellite Events at the MoDELS 2005 Conference*, pages 128–138. Springer, 2005. (ISBN 3 540 31780 5).

[8] Philippe Kruchten. *The Rational Unified Proces - An Introduction*. Addison-Wesley, 1999. (ISBN 0 201 70710 1).

[9] Daniel L. Moody and Graeme G. Shanks. What makes a good data model? evaluating the quality of entity relationship models. In *ER '94: Proceedings of the 13th International Conference on the Entity-Relationship Approach*, pages 94–111, London, UK, 1994. Springer-Verlag. (3-540-58786-1).

[10] NEderlandse Software Metrieken Gebruikers Associatie (NESMA). *FPA-Casus "hotel" — Voor de toepassing van functiepuntanalyse — Een casus voor de praktijk — Versie 2.0*, 1998.

[11] NEderlandse Software Metrieken Gebruikers Associatie (NESMA). *Software engineering — NESMA functional size measurement method version 2.1 — Definitions and counting guidelines for the application of Function Point Analysis. International Standard ISO/IEC 24570:2005(E)*, 2005.

[12] NEderlandse Software Metrieken Gebruikers Associatie (NESMA). *FPA toegepast bij UML/Use Cases — Versie 1.0*, 2008.

[13] NEderlandse Software Metrieken Gebruikers Associatie (NESMA). *FPA volgens NESMA en IFPUG; de actuele stand van zaken*, 2008.

[14] Object Management Group (OMG). *Meta Object Facility (MOF) Specification, version 1.4*, 2002.

Appendix A

Complexity of files and transactions in the hotel reservation system

Complexity of files

In Chapter 3 described the hotel reservation system. To perform a detailed count, the complexity of these files has to be determined. In order to do so, we have to identify the record types in each file and the data elements in those record types. We have done this for each file in the system. In this appendix the complexity determination is specified in more detail.

Reservation

This internal logical file consists of only one record type, namely reservation. In this record the reservation number, customer, room type, arrival date, length of stay and the number of active room registrations are recorded. In total this record contains six data elements. According to Table 2.3 this is an ILF of low complexity.

Customer

This is also an internal logical file which consists of only the record type customer. In this record the customer number, the name of the customer, the address of the customer, the city, country code, phone number, email address and language are recorded. In total this record contains eight data elements. According to Table 2.3 this is an ILF of low complexity.

Room and room type

This is an internal logical file which consists of two record types, namely room and room type. In the record room the data elements room number and room type are recorded. In the record room type the name of the room type, the price for a room of that room type, a description of the room type in English, a description of the room type in Dutch, a description in French and a description in German are recorded. In total these two records contain eight data elements. According to Table 2.3 this is an ILF of low complexity.

Room registration

This, again, is an internal logical file which consists of only a single record type, room registration. In this record the registration number, the room type of the registered room,

the room number, the arrival date, the intended duration of the stay, the customer and the departure date are recorded. In total this record contains seven data elements. According to Table 2.3 this is an ILF of low complexity.

Bill
This is an external interface file which consists of a record type bill. In this record the data elements invoice number, amount to be paid and the number of the room registration to which the bill is attached are recorded. In total this record contains three data elements. According to Table 2.4 this is an ILF of low complexity.

Hotel
This is a FPA table which is maintained externally, from which general information about the hotel can be retrieved. In this FPA table the name, address, city, country, phone number, fax number, email and the manager of the hotel are recorded.

Country
This is also an externally maintained FPA table. It is used for converting between country codes and country names. Therefore, the table consists of only two data element types, namely country name and country code.

The FPA table EIF are counted altogether as one EIF. There are two FPA tables EIF which have 10 data element types in total. According to Table 2.4 the complexity of the EIF counted for the FPA tables EIF is low.

Complexity of transactions
In order to perform a detailed function point count we also need to determine the complexity of the transactions in the hotel reservation system. In order to do so, we need to know which files are referenced by the transactions and which data elements crossed the system boundary during the transactions. Data elements that are referenced to but that do not cross the system boundary should not be counted when determining the complexity of a transaction. Also, if multiple data elements, for example if the data elements surname and name are used for storing a persons name, these should be counted as one data element crossing the system boundary for a transaction in which a person's name is displayed. In this section of this appendix we will specify the determination of the complexity of the transactions in more detail.

Request reservation
File: Reservation
Data elements: desired room type, planned arrival date, intended length of the stay

File: Customer
Data elements: name, address, city, country, phone number, email address, language

2 files and 10 data elements: EI of average complexity

Lookup room types
File: Room and room type
Data elements: name, price, description in certain language

1 file and 3 data elements: EO of low complexity

Check room availability
File: Room registration
Data elements: room type, arrival date, length of stay

File: Reservation
Data elements: room type, arrival date, length of stay

2 files and 6 data elements: EO of average complexity

Confirm reservation
File: Customer
Data elements: customer number, name, address, city, country, phone number, email address, language

File: Reservation
Data elements: arrival date, length of stay

File: Room and room type
Data elements: name, price, description in preferred language

FPA Table: Hotel
Data elements: name, address, city, country, phone number, fax number, email address, name of manager

3 files and 21 data elements: EO of high complexity

Lookup reservation
File: Reservation
Data elements: reservation number, room type, arrival date, length of stay, number of registered rooms

1 file and 6 data elements: EO of low complexity

Alter reservation
File: Reservation

Data elements: room type, arrival date, length of stay

File: Customer
Data elements: name, address, city, country, phone number, email, language

2 files and 10 data elements: EI of average complexity

Cancel reservation
File: Reservation
Data elements: reservation number, room type, arrival date, length of stay, number of room registrations

File: Customer
Data elements: customer number, name, address, city, country

2 files and 10 data elements: EI of average complexity

Lookup reservations arriving today
File: Reservation
Data elements: reservation number, room type, arrival date, length of stay

1 file and 4 data elements: EO of low complexity

Register with reservation
File: Room registration
Data elements: room type, room number, arrival date, duration of stay, customer number

1 file and 5 data elements: EI of low complexity

Alter customer details
File: Customer
Data elements: name, address, city, country, phone number, language, email

1 file and 7 data elements: EI of low complexity

Register without reservation
File: Room registration
Data elements: room type, room number, arrival date, duration of stay

File: Customer
Data elements: customer name, address, city, country code, phone number, language, email address

2 files and 11 data elements: EI of average complexity

Unregister
File: Room registration
Data elements: room number, departure date

1 file and 2 data elements: EI of low complexity

Request cost overview
File: Bill
Data elements: invoice number, amount

File: Room registration
Data elements: registration number, room type, room number, arrival date, duration of stay, departure date

File: Customer
Data elements: customer number, name, address, city, country

3 files and 13 data elements: EO of average complexity

Add room
File: Room and room type
Data elements: room number, room type

1 file and 2 data elements: EI of low complexity

Lookup room
File: Room and room type
Data elements: room number, room type

1 file and 2 data elements: EQ of low complexity

Alter room info
File: Room and room type
Data elements: room number, room type

1 file and 2 data elements: EI of low complexity

Check whether room is in use
File: Room registration
Data elements: room number, arrival date, duration of stay

1 file and 3 data elements: EO of low complexity

Remove room
File: Room and room type
Data elements: room number

1 file and 1 data element: EI of low complexity

Add room type
File: Room and room type
Data elements: name, price, description in English, Dutch, French and German

1 file and 6 data elements: EI of low complexity

Lookup room type
File: Room and room type
Data elements: name, price, description in English, Dutch, French and German

1 file and 6 data elements: EQ of low complexity

Alter room type
File: Room and room type
Data elements: name, price, description in English, Dutch, French and German

1 file and 6 data elements: EI of low complexity

Check whether room type is in use
File: Room registration
Data elements: room type, arrival date, duration of stay

File: Reservation
Data elements: room type, arrival date, duration of stay

2 file and 6 data elements: EO of average complexity

Remove room type
File: Room and room type
Data elements: name

1 file and 1 data elements: EI of low complexity

Appendix B

Pseudocode for the various mappings

In this appendix the pseudocode for the various mappings as described in Chapter 3 is given. This pseudocode is a simplified depiction of the implementation of the mappings. For example, in the detailed count complexities of the functions have to be determined. This is done by checking the conditions of the functions and comparing them to the NESMA tables given in Chapter 2. Writing out these checks in pseudocode detailedly, would make the code much more verbose than it is now and could be the cause of losing the overview of what is happening. Another example is the SFRL2UML mapping in which we mention which element is created with which stereotype, while in reality we would create a stereotype element and make references from other elements to this element.

B.1 SFRL to indicative function point count

```
create new IndicativeCount i

i.systemName = name of system
i.numILF_low = number of ILFs in System
i.numEIF_low = number of EIFs in System

functionpoints = 0

functionpoints = functionpoints + (number of ILFs in System * 35)
functionpoints = functionpoints + (number of EIFs in System * 15)

if (number of FPA tables ILF in system > 0) then
    functionpoints = functionpoints + 35
    i.FPA_ILF_points = 35

if (number of FPA tables EIF in system > 0) then
    functionpoints = functionpoints + 15
    i.FPA_EIF_points = 15
```

```
i.functionPointCount = functionpoints
```

Listing B.1: Pseudocode for performing an indicative function point count

B.2 SFRL to estimated function point count

```
create new EstimatedCount e

e.systemName = name of system
e.numILF_low = number of ILFs in System
e.numEIF_low = number of EIFs in System
e.numEI_avg = number of EIs in System
e.numEO_avg = number of EOs in System
e.numEQ_avg = number of EQs in System

functionpoints = 0

// files are of low complexity in estimated count
functionpoints = functionpoints + (number of ILFs in System * 7)
functionpoints = functionpoints + (number of EIFs in System * 5)

// transactions are of average complexity in estimated count
functionpoints = functionpoints + (number of EIs in System * 4)
functionpoints = functionpoints + (number of EOs in System * 5)
functionpoints = functionpoints + (number of EQs in System * 4)

if (number of FPA tables ILF in System > 0) then
    // 1 ILF + 1 EI + 1 EO + 1 EQ
    functionpoints = functionpoints + 7 + 4 + 5 + 4
    e.FPA_ILF_points = 7 + 4 + 5 + 4

if (number of FPA tables EIF in System > 0) then
    // 1 EIF
    functionpoints = functionpoints + 5
    e.FPA_EIF_points = 5

e.functionPointCount = functionpoints
```

Listing B.2: Pseudocode for performing an estimated function point count

B.3 SFRL to detailed function point count

```
create new DetailedCount d

d.systemName = name of system
d.numILF_low = number of ILFs with low complexity in System
d.numEIF_low = number of EIFs with low complexity  in System
```

```
d.numILF_avg = number of ILFs with avg complexity in System
d.numEIF_avg = number of EIFs with avg complexity  in System
d.numILF_high = number of ILFs with high complexity in System
d.numEIF_high = number of EIFs with high complexity  in System

d.numEI_low = number of EIs with low complexity in System
d.numEO_low = number of EOs with low complexity in System
d.numEQ_low = number of EQs with low complexity in System
d.numEI_avg = number of EIs with avg complexity in System
d.numEO_avg = number of EOs with avg complexity in System
d.numEQ_avg = number of EQs with avg complexity in System
d.numEI_high = number of EIs with high complexity in System
d.numEO_high = number of EOs with high complexity in System
d.numEQ_high = number of EQs with high complexity in System

functionpoints = 0
FPA_ILF_points = 0
FPA_EIF_points = 0

//complexity of ILF is determined according to Table 2.3
for all ILFs in System with low complexity do
    functionpoints = functionpoints + 7
for all ILFs in System with average complexity do
    functionpoints = functionpoints + 10
for all ILFs in System with high complexity do
    functionpoints = functionpoints + 15

//complexity of EIF is determined according to Table 2.4
for all EIFs in System with low complexity do
    functionpoints = functionpoints + 5
for all EIFs in System with average complexity do
    functionpoints = functionpoints + 7
for all EIFs in System with high complexity do
    functionpoints = functionpoints + 10

//complexity of EI is determined according to Table 2.5
for all EIs in System with low complexity do
    functionpoints = functionpoints + 3
for all EIs in System with average complexity do
    functionpoints = functionpoints + 4
for all EIs in System with high complexity do
    functionpoints = functionpoints + 6

//complexity of EO is determined according to Table 2.6
for all EOs in System with low complexity do
    functionpoints = functionpoints + 4
for all EOs in System with average complexity do
    functionpoints = functionpoints + 5
for all EOs in System with high complexity do
```

```
    functionpoints = functionpoints + 7

//complexity of EQ is determined according to Tables 2.5 and 2.6
//whichever part (input or output) has higher complexity counts
for all EQs in System with low complexity do
    functionpoints = functionpoints + 3
for all EQs in System with average complexity do
    functionpoints = functionpoints + 4
for all EQs in System with high complexity do
    functionpoints = functionpoints + 6

//complexity of ILF is determined according to Table 2.3
//number of FPA tables ILF are counted as the number of RETs
//total amount of DETs are counted as the number of DETs
if (complexity of FPA tables ILF in System is low) then
    FPA_ILF_points = FPA_ILF_points + 7
if (complexity of FPA tables ILF in System is average) then
    FPA_ILF_points = FPA_ILF_points + 10
if (complexity of FPA tables ILF in System is high) then
    FPA_ILF_points = FPA_ILF_points + 15

//1 EI, 1 EO and 1 EQ is counted for all FPA tables ILF together
//complexity is determined according to tables 2.5 and 2.6
//number of FPA tables ILF counts as the number of FTR
//number of DETs in the FPA tables counts as the number of DETs
if (complexity of extra EI for FPA tables ILF is low) then
    FPA_ILF_points = FPA_ILF_points + 3
if (complexity of extra EI for FPA tables ILF is average) then
    FPA_ILF_points = FPA_ILF_points + 4
if (complexity of extra EI for FPA tables ILF is high) then
    FPA_ILF_points = FPA_ILF_points + 6

if (complexity of extra EO for FPA tables ILF is low) then
    FPA_ILF_points = FPA_ILF_points + 4
if (complexity of extra EO for FPA tables ILF is average) then
    FPA_ILF_points = FPA_ILF_points + 5
if (complexity of extra EO for FPA tables ILF is high) then
    FPA_ILF_points = FPA_ILF_points + 7

if (complexity of extra EQ for FPA tables ILF is low) then
    FPA_ILF_points = FPA_ILF_points + 3
if (complexity of extra EQ for FPA tables ILF is average) then
    FPA_ILF_points = FPA_ILF_points + 4
if (complexity of extra EQ for FPA tables ILF is high) then
    FPA_ILF_points = FPA_ILF_points + 6

functionpoints = functionpoints + FPA_ILF_points
d.FPA_ILF_points = FPA_ILF_points
```

```
//complexity of EIF is determined according to Table 2.4
//number of FPA tables EIF are counted as the number of RETs
//total amount of DETs are counted as the number of DETs
if (complexity of FPA tables EIF in System is low) then
    FPA_EIF_points = FPA_EIF_points + 5
if (complexity of FPA tables EIF in System is average) then
    FPA_EIF_points = FPA_EIF_points + 7
if (complexity of FPA tables EIF in System is high) then
    FPA_EIF_points = FPA_EIF_points + 10

functionpoints = functionpoints + FPA_EIF_points
d.FPA_EIF_points = FPA_EIF_points
```

Listing B.3: Pseudocode for performing a detailed function point count

B.4 SFRL to UML components

```
create a model with the same name as the system in SFRL
create UML components package hierarchy
for all EIs in SFRL
    create a UseCase
        with the name [EI.name]
        with stereotype <<input>>
        in the package /Requirements/Use Case Model/Transactions
    create an Operation
        with the name [EI.name]
        with stereotype <<input>>
        in system interfaces derived from a supported UserGoal
        for all files referenced by EI as input
            create parameter in operation
                with the name in_+[File.name]
                with direction kind in
            create Dependency
                from parameter
                to data element
            create Class in System Interface Package
                with stereotype <<info type>>
                with name [File.name]
        for all files referenced by EI as output
            create parameter in operation
                with the name out_+[File.name]
                with direction kind out
            create Dependency
                from parameter
                to data element
            create Class in System Interface Package
                with stereotype <<info type>>
                with name [File.name]
for all EOs in SFRL
```

```
        create a UseCase
            with the name [EO.name]
            with stereotype <<output>>
            in the package /Requirements/Use Case Model/Transactions
        create an Operation
            with the name [EO.name]
            with stereotype <<output>>
            in system interfaces derived from a supported UserGoal
            for all files referenced by EO as input
                create parameter in operation
                    with the name in_+[File.name]
                    with direction kind in
                create Dependency
                    from parameter
                    to data element
                create Class in System Interface Package
                    with stereotype <<info type>>
                    with name [File.name]
            for all files referenced by EO as output
                create parameter in operation
                    with the name out_+[File.name]
                    with direction kind out
                create Dependency
                    from parameter
                    to data element
                create Class in System Interface Package
                    with stereotype <<info type>>
                    with name [File.name]
    for all EQs in SFRL
        create a UseCase
            with the name [EQ.name]
            with stereotype <<inquiry>>
            in the package /Requirements/Use Case Model/Transactions
        create an Operation
            with the name [EQ.name]
            with stereotype <<inquiry>>
            in system interfaces derived from a supported UserGoal
            for all files referenced by EQ as input
                create parameter in operation
                    with the name in_+[File.name]
                    with direction kind in
                create Dependency
                    from parameter
                    to data element
                create Class in System Interface Package
                    with stereotype <<info type>>
                    with name [File.name]
            for all files referenced by EQ as output
                create parameter in operation
```

```
                with the name out_+[File.name]
                with direction kind out
            create Dependency
                from parameter
                to data element
            create Class in System Interface Package
                with stereotype <<info type>>
                with name [File.name]
for all users in SFRL
    create an Actor
        with the name [User.name]
        in the package /Requirements/Use Case Model
for all usergoals in SFRL
    create a UseCase
        with the name [UserGoal.name]
        in the package /Requirements/Use Case Model
    create a Class
        with the name I+[UserGoal.name]
        with the stereotype <<interface type>>
        in the package /Specification/Interface Specification
            /System Interfaces
    determine supporting transactions and
        create Dependencies in UML with stereotype <<include>>
for all business goals in SFRL
    create a UseCase
        with the name [BusinessGoal.name]
        in the package /Requirements/Use Case Model
            /Business Goals
for all ILFs in SFRL
    create a Class
        with the name [ILF.name]
        with stereotype <<FPA File>>
        in package /Requirements/Business Concept Model
    for all records in ILF
        create a Class
            with the name [Record.name]
            with stereotype <<concept>>
            in the class created in the Business Concept Model
        create a Class
            with the name [Record.name]
            with stereotype <<type>>
            in package /Requirements/Business Type Model
        for all data elements in Record
            create Attribute
                with the name [DET.name]
                in the type created in the Business Type Model
for all EIFs in SFRL
    create a Class
        with the name [EIF.name]
```

```
            with stereotype <<FPA File>>
            in package /Requirements/Business Concept Model
        create a Class
            with the name [EIF.name]+System
            with stereotype <<comp spec>>
            in the package /Specification/Component Specification
        for all records in EIF
            create a Class
                with the name [Record.name]
                with stereotype <<concept>>
                in the class created in the Business Concept Model
            create a Class
                with the name [Record.name]
                with stereotype <<external type>>
                in package /Requirements/Business Type Model
            for all data elements in Record
                create Attribute
                    with the name [DET.name]
                    in the type created in the Business Type Model
    for all FPA tables ILF in SFRL
        create a Class
            with the name [FPATableILF.name]
            with stereotype <<concept>>
            in package /Requirements/Business Concept Model
        create a Class
            with the name [FPATableILF.name]
            with stereotypes <<type>>
            in package /Requirements/Business Type Model
        for all data elements in FPATableILF
            create Attribute
                with the name [DET.name]
                in the type created in the Business Type Model
    for all FPA tables EIF in SFRL
        create a Class
            with the name [FPATableEIF.name]
            with stereotypes <<concept>>
            in package /Requirements/Business Concept Model
        create a Class
            with the name [FPATableEIF.name]
            with stereotypes <<external type>>
            in package /Requirements/Business Type Model
        for all data elements in FPATableEIF
            create Attribute
                with the name [DET.name]
                in the type created in the Business Type Model
    for all IndependentDETs in SFRL
        for all System Interfaces which use IndependentDET
            create Class in System Interface Package
                with name FPA_Independent_DETs
```

```
            with stereotype <<info type>>
```

Listing B.4: Pseudocode for generation a UML components specification from SFRL

B.5 UML components to SFRL

```
create a System in SFRL with the same name as the model in UML

for all Classes in Business Goals Package
    create BusinessGoal
        with the name [UseCase.name]
for all Actors in Use Case Model
    create a User in SFRL
        with the name [Actor.name]
for all UseCases in Use Case Model
    create UserGoal in SFRL
        with the name [UseCase.name]
        with associations to Users of UserGoal
        with association to supported BusinessGoal
for all UseCases in Transactions Package
    with stereotype <<input>>
    create EI in SFRL
        with the name [UseCase.name]
        with associations to supported UserGoals
for all UseCases in Transactions Package
    with stereotype <<output>>
    create EO in SFRL
        with the name [UseCase.name]
        with associations to supported UserGoals
for all UseCases in Transactions Package
    with stereotype <<inquiry>>
    create EQ in SFRL
        with the name [UseCase.name]
        with associations to supported UserGoals
if stereotype <<FPA File>> exists
    for all Classes in Business Type Model
            with stereotype <<FPA File>>
        with <<concept>> Classes of which <<type>> Classes
            exist in Business Type Model
        create ILF in SFRL
            with the name [Class.name]
    for all Classes in Business Type Model
        with stereotype <<FPA File>>
        with <<concept>> Classes of which <<external type>>
            Classes exist in Business Type Model
        create EIF in SFRL
            with the name [Class.name]
    for all Classes in FPA Files in Business Concept Model
        with stereotype <<concept>>
```

```
            with equivalent in Business Type Model
                with stereotype <<type>>
            create Record
                in ILF with dependency to equivalent concept Class
                    with the name [Class.name]
                    for all Attributes in Class
                        create DET in Record
                            with the name [Attribute.name]
        for all Classes in FPA Files in Business Concept Model
            with stereotype <<concept>>
            with equivalent in Business Type Model
                with stereotype <<external type>>
            create Record
                in EIF with dependency to equivalent concept Class
                    with the name [Class.name]
                    for all Attributes in Class
                    create DET in Record
                        with the name [Attribute.name]
        for all Classes in Business Concept Model
            with stereotype <<concept>>
            with equivalent in Business Type Model
                with stereotype <<type>>
            create FPATableILF
                with the name [Class.name]
                for all Attributes in Class
                    create DET in FPATableILF
                        with the name [Attribute.name]
        for all Classes in Business Concept Model
            with stereotype <<concept>>
            with equivalent in Business Type Model
                with stereotype <<external type>>
            create FPATableEIF
                with the name [Class.name]
                for all Attributes in Class
                    create DET in FPATableEIF
                        with the name [Attribute.name]
        for all Attributes in Class FPA_Independent_DETs
        in Interface Data Types Package
            with stereotype <<infotype>>
            create IndependentDET
                with the name [Attribute.name]
    else
        for all Classes in Business Concept Model
            with equivalent in Business Type Model
            create ILF
                with the name [Class.name]
        for all Classes in Business Concept Model
            without equivalent in Business Type Model
            create EIF
```

```
            with the name [Class.name]
```

Listing B.5: Pseudocode for extracting information from a UML components specification and recording it in SFRL