Analysis of requirements incompleteness using metamodel specification

Ao Li

University of Tampere
School of Information Sciences
Computer Science / Software Development M.Sc. thesis
Supervisor: Zheying Zhang
June 2015

University of Tampere
School of Information Sciences
Computer Science / Software Development
Ao Li: Analysis of requirements incompleteness using metamodel
M.Sc. thesis, 52 pages and 3 index and appendix pages
June 2015

Abstract.

Incompleteness of requirements has been treated as a huge challenge in software development projects. Since it is hard to obtain all required information before software design and implementation starts, the software modeling process may start with an incomplete requirements specification. In order to help analyzing the incompleteness in requirements, I propose a metamodel approach for detecting the missing requirements that are needed for constructing conceptual models for a software system, and implement it in MetaEdit+. The detected missing information in a conceptual model is reported in natural language, which is easy to understand. Furthermore, the conceptual modelers can identify the potential problems indicated by the report to analyze and update the model. The contribution of my thesis is twofold, i.e. analyzing the link between business rules and the ER models, and implementing a method to automatically detect and show the incompleteness in ER models.

Keywords: Requirements incompleteness, business rules, metamodel, MetaEdit+.

Contents

1. Introduction

Requirements are uncertain, and the uncertainties are increasing with fast changing markets [Ebert and De Man, 2005]. Requirements uncertainty means that "within a software system, requirements are not known until it is practically used" [Parnas, 1979]. Uncertainty includes the unclear information on project goals, feasibility, cost, and duration of implementing these alternatives, as well as the future changes in stakeholders' goals, business context and technological environments [Letier et al., 2014]. Uncertain requirements can be caused by many reasons, such as insufficient management of changing requirements [Ebert and De Man, 2005], or difficulties to predict all of the details required before a project start [Zhang et al., 2014].

As one of the early symptoms of uncertainty, requirements incompleteness will lead to problems from different perspectives, such as insufficient acceptance and satisfaction from users, underestimated costs and schedules, ambiguous verification and more safety incidents [Wiegers, 1999]. Therefore, detecting incomplete requirements becomes an important issue in software development and considerable research effort has been put on developing methodologies to improve the completeness, consistency, and accuracy of requirements in a requirements engineering (RE) process [check examples in Carson, 1998; Zowghi and Gervasi, 2006; Zhang et al., 2014].

In a software project, requirements are specified on different abstraction levels. As the high abstraction level requirements, business rules are part of rich sources of requirements and also a volatile part of a software system [Wan-Kadir and Loucopoulos, 2003]. In order to explain the link between business rules and the software system structure, the Manchester Business Rules Management (MBRM) mapping metamodel method has been introduced to connect business rules elicitation with software modelling, which will keep traceability on link modeling process and reducing the effort on requirements changes management [Wan-Kadir and Loucopoulos, 2003].

On the basis of my understanding of requirements incompleteness, I further study the techniques which can be used to detect incomplete requirements during the conceptual modeling process. Thanish, et al. [2013] presented a way of representing some of that incompleteness in the conceptual model. A list of incompleteness problems are discussed based on the Enhanced Entity-Relationship (EER) model. According to the list of problems, a metamodel approach for detecting the unknown requirements is proposed by Zhang et al [2013]. A metamodel defines the components of a conceptual model, process, or system. By building a database with the information of metamodel

elements, some of the unknown information can be checked automatically by SQL code and reported in natural language.

Based on the MBRM framework and the metamodeling technique, my work is to implement and to demonstrate the existing approach in a specific metamodeling tool, i.e. MetaEdit+. MetaEdit+ [Kelly et al., 1998] provides an explicit description of MetaEdit+ function and architecture as well as definition of GOPRR metamodeling language. A link model which links business rules and ER model elements will be proposed and analyzed, and metamodeling techniques will be applied to detect and capture the incomplete requirements in ER models.

By using MetaEdit+, one can build different generators to report the incompleteness detection and feedback. Based on the generated results, all incomplete information can be shown in a report and analyzed with their related business rules. By analyzing the results, one can reduce the uncertainties caused by incomplete information on requirements and identify the missing business rules. The contribution of my thesis is twofold. First, I detected and analyzed the 12 incompleteness problems which are provided by Thanish, et al. [2013], by using the metamodeling techniques and generator function in MetaEdit+. The approach is demonstrated using an example from a meeting scheduling process. Second, I built the link between business rules and ER model elements based on the MBRM link model, which shows the traceability of ER model incompleteness problems and updates the business rules at the same time. Some of the incompleteness can be found and updated in both model and business rules, while some incomplete information cannot be captured because of the limitation of the metamodel concepts.

The content of the rest of this thesis is as follows. Chapter 2 introduces the basic definitions of business rules and requirements uncertainty, including the relationship between the uncertainty and incompleteness, and root causes of incomplete requirements. Chapter 3 introduces the concepts of metamodel and the MBRM mapping metamodeling methodology in detail, and discusses how this framework is applied to deal with incomplete requirements. Introduction of the MetaEdit+ tool and the GOPRR metamodeling language will be described in the Chapter 4. Chapter 5 addresses the possible incompleteness in an ER model in a list, together with examples. Chapter 6 presents the implementation in MetaEdit+ about the detection and reporting of incomplete requirements. Chapters 7 and 8 includes further discussion and the conclusion.

2. Requirements and uncertainty

Frederick Brooks [1987] eloquently stated the critical role of requirements in a software project as “the hardest single part of building a software system is deciding precisely what to build and no other part of the conceptual work is as difficult as establishing the detailed technical requirements”. However, requirements uncertainty is an inevitable problem in the RE process, and in order to solve this problem, we need to figure out what the requirements uncertainty is and what causes the uncertainty.

2.1. Requirements

A requirement is a statement of a system service or constraint which is specified for the implementation activity in a software development process [Kotonya and Sommerville, 1998]. The statements can be a functional requirement or a nonfunctional requirement. A functional requirement should describe the behavior of a system, like what the system is required to operate under a specific circumstance. A non-functional requirement specifies the type and property of a system, as well as the constraints of system operation and details of computation in the software process [Kotonya and Sommerville, 1998]. In other words, the requirements contain both the specific conditions to the system and properties which make the system suitable and even enjoyable to users [Wiegers and Beatty, 2013]. The IEEE standard [IEEE std. 610.12, 1990] defines the requirements from both users and system perspectives. For users, a requirement is the specification to solve a problem or achieve an objective; while for a system, a requirement is to satisfy a contract, standard, specification or other formally imposed documents.

Requirements are often classified into functional requirements, quality requirements and constraints [Pohl and Rupp, 2011]. A quality requirement concerns aspects that are not covered by functional requirements, such as performance, availability, dependability, scalability, or portability of a system. Requirements of this type are frequently classified as non-functional requirements. A constraint is a requirement that limits the solution space beyond what is necessary for meeting the given functional requirements and quality requirements.

In addition, there are distinct classifications of requirements from different perspectives. For example, a three-level requirements model has been defined in [Wiegers and Beatty, 2013]. As shown in Figure 1, requirements can be distinguished between three abstraction levels, i.e. business requirements, user requirements, and functional requirements. Business requirements specify high-level business objectives of an organization. In this level, the reason of implementing the system needs to be

found, or in other words, the business benefits which the organization expects to achieve is the key point of this part [Wiegers and Beatty, 2013]. The business requirements can be written into a project vision and scope document.

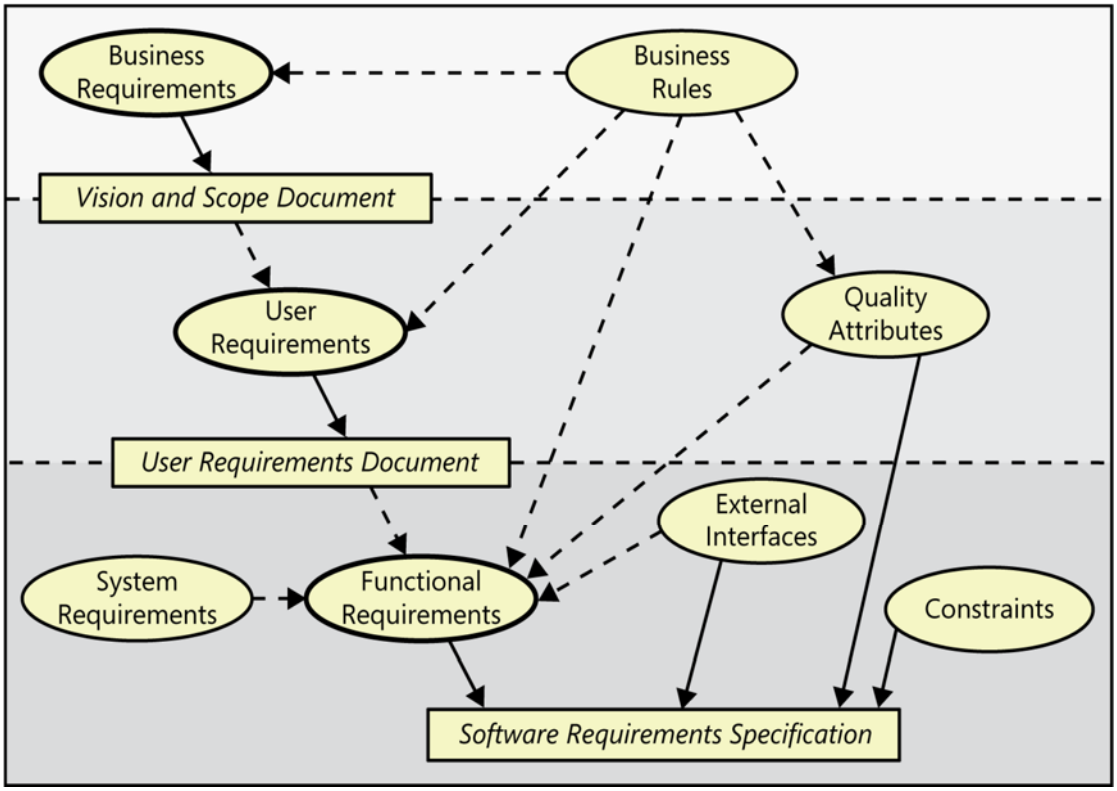

Figure 1. Relationships among several types of requirements [Wiegers and Beatty, 2013].

User requirements elaborate on the business requirements by describing goals or tasks of users' performance and product attributes or characteristics to satisfy the users. In this level, what actual users need to get from the help of the product is the main question. The user requirements are often documented in use cases and user stories [Wiegers and Beatty, 2013]. A use case describes a sequence of interactions between a system and an external actor that results in the actor being able to achieve some outcome of value [Wiegers, 2014]. Use cases should describe user goals, the user's view of the system and a set of task-related activities. A use case is always written in the form of a verb followed by an object. For example, an online bookstore has a use case of *update customer profile* [Cohn, 2010]. Each use case has a corresponding user story which is a short, simple description of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system [Cohn, 2010]. With the online bookstore example above, the relevant user story of "update customer profile" can be stated like *as a customer, I want to update my customer profile so that future purchases are billed to a new credit card number* [Cohn, 2010].

Derived from user requirements, functional requirements represent what the system should do under specific conditions [Wiegers and Beatty, 2013]. In this level, implementation details have been defined to accomplish both the user requirements and business requirements. Besides, system requirements, external interfaces and quality attributes provide the necessary addition to functional requirements.

System requirements describe functions and qualities which the system implements or possesses. In addition, system requirements support the rest of development activities with the complete information [Pohl and Rupp, 2011].

According to Richard Thayer [2002], external interface requirements specify hardware, software, or database elements with which a system or component must interface. These requirements include the information of user interfaces, software interfaces, hardware interfaces, and communication interfaces. The interface can describe the connections between software and the universe. For example, *a mobile application should send the check image to the bank after someone photograph the check they are depositing*, which states the interface between the mobile application and the banking system [Wiegers and Beatty, 2013].

A quality attribute pertains to a quality concern that is not covered by functional requirements, which is responsible for description of system services and performance [Pohl and Rupp, 2011]. General software quality attributes include scalability, security, performance, and reliability [Gorton & Ernst, 2014]. An example of a scalability requirement for a computer company's information technology infrastructure is like *it must be possible to scale the deployment from an initial 100 geographically dispersed user desktops to 10,000 without an increase in effort/cost for installation and configuration* [Gorton & Ernst, 2014].

2.2. **Business rules**

In Figure 1, the business rules are on the same abstraction level as business requirements, and form an important source of requirements. Usually, business rules are elicited in the first stage of a requirements engineering process and they have been specified by different stakeholders in different ways. In general, business rules are rich source of requirements and system must obey the rules [Wiegers and Beatty, 2013]. They are used to represent both user requirements and conditions to which the system should conform [Wan-Kadir and Loucopoulos, 2003]. From an enterprise perspective, business rules are statements of goals, policies, constraints on an enterprise's way of doing business [Rosca et al., 1999] or statements about how the business is done, i.e. about guidelines and restrictions with respect to states and processes in an organization

[Herbst, 1996]. In detail, Halle [1994] suggests that depending on whom you ask, business rules may encompass some or all of relationship verbs, mathematical calculations, inference rules, step-by-step instructions, database constraints, business goals and policies, and business definitions.

When a client says that some activity can only be done in some specific condition and only by some specific person, it is a business rule. For example, *a new client must pay 30 percent of the estimated consulting fee and travel expenses in advance* in a lawyer company [Wiegers and Beatty, 2013].

Business rules can be defined in five classes, and they are facts, constraints, action enablers, inferences and computations, as shown in Figure 2.

Facts describe associations or relationships between important business terms [Wiegers and Beatty, 2013]. For example, *every book in the library should have a unique bar code*. Unnecessary facts can *bog down* the business analysis so we should pay more attention on those facts which are in scope for the project and make the system events more clear [Wiegers and Beatty, 2013].

Constraints, as the name suggests, define the specific actions done or inhabited by certain people. For instance, *a library card applicant must be more than 16 years old*. Certain business rules use the constraints to restrict the way that the business operates.

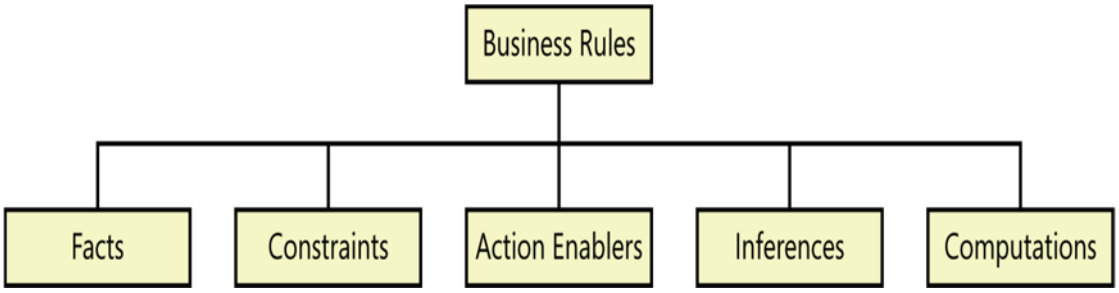

Figure 2. A simple business rule taxonomy [Wiegers and Beatty, 2013].

An action enabler triggers some activities if specific conditions are true in an “if-then” form to describe a requirement scenario [Wiegers and Beatty, 2013]. The “if” clause acts as an enabler while the “then” clause indicates the follow-up activity. For example, *if a reader ordered one book from an author who also has many other books, system shall recommend them to the user* [Wiegers and Beatty, 2013], which stimulates impulse purchases after a customer decide to buy a specific product.

An inference is similar to "if-then" format as an action enabler, but different from the action enabler, the “then” clause of an inference simply provides a piece of knowledge, not an action to be taken [Wiegers and Beatty, 2013]. For instance, *if a*

reader did not return his/her book in time, then the account is considered as "delinquent".

A computation transforms existing data into new data by using specific mathematical formulas or algorithms. For example, book selling in the bookstore of a library may have discounts *by 10 percent for orders of 6 to 10 units, by 20 percent for orders for orders of 11 to 20 units* [Wiegers and Beatty, 2013].

However, the common requirements identification and specification process is dominated by technical concerns rather than business concerns [Ebert and De Man, 2005]. In other words, a requirements elicitation process sometimes ignore the constraints regarding the business process and many software project problems are caused by insufficient identification of business rules. Therefore, an explicit stage where business needs are identified by the stakeholders is necessary [Ebert and De Man, 2005]. Several approaches which focus on business rules elicitation and analysis have been proposed, as presented in Chapter 3.

2.3. Uncertainty and incompleteness in requirements

Badly written requirements are easily identified in many software requirements specifications and no matter how much effort is put on requirement analyzing, reviewing, and refining, the requirements are rarely widely accepted by heterogeneous groups of stakeholders [Wiegers, 1999]. Various problems are caused by textual requirements written in natural language [Sommerville, 2011]. One drawback of natural language, i.e. unclearness and the lack of clarity, makes it difficult to specify requirements in a precise and unambiguous way, and makes the document wordy and difficult to read at the same time. Functional requirements, non-functional requirements, and other design information could be unclearly distinguished with natural language, which will raise the confusion in requirements analysis. Moreover, several different requirements may be expressed together as a single requirement, which forms a risk of requirements "amalgamation" [Sommerville, 2011]. Although the importance of requirements quality has been recognized by the software development team, some common requirements problems still exist in the RE process, i.e. insufficiency in requirements completeness and accuracy [Wiegers, 1999].

2.3.1. Requirements uncertainty

As the inability to determine the true state of affairs of a system or things that are not known, or known only imprecisely [McManus & Hastings, 2005], uncertainty is a common problem in product development. Many different views and numerous classifications of uncertainty have been proposed in this area. A product development

process uncertainty can be divided into two categories [Chalupnik et al., 2009], i.e. exogenous process uncertainty (or external process uncertainty) and the endogenous process uncertainty (or internal process uncertainty).

The exogenous uncertainty indicates the uncertainty in the product development process environment, such as the organizational change, the unstable markets, user expectations changes, or the evolution of the political and cultural contexts of the company [Bstieler, 2005].

The endogenous process uncertainty is caused by technology novelty and process complexity. Technology novelty has two components: insufficient understanding of new product technology and uncertainty on process technology. Process complexity reflects the difficulty in understanding the process objectives, the newness of those objectives to the company, and the degree of interdependence among product elements [Chalupnik et al., 2009]. Both of these uncertainties lead to a complex and competitive environment and compliance with various stakeholders and regulatory bodies complex and changing requirements [Chalupnik et al., 2009].

In a software development process, uncertain requirements can be shown as both exogenous and endogenous process uncertainty because they always volatile with the development of rapid changing market [Ebert and De Man, 2005] and the attitude of keep modifying requirements. Moreover, unresolved decisions, insufficient understanding among stakeholders, etc., can all lead to uncertainties in requirements [Salay et al., 2012]. Since it is not always possible to provide a complete and consistent description [Thanish et al., 2013] [Zhang et al., 2014] of requirements before the modeling and the coding starts, uncertainty should be admitted, represented and solved if it has been detected [Bonarini, 2010].

2.3.2. Causes of software requirements uncertainty

Five different root causes of requirements uncertainty have been discussed deeply, and they are vague product vision and strategy, absence of stakeholders and misunderstanding among them, unknown project dependencies, not thoroughly evaluated business case and insufficient specification and analysis of requirements [Ebert and De Man, 2005], as shown in Figure 3.

A product vision describes what the product is about and what it ultimately could become to achieve the business objectives [Wiegers and Beatty, 2013]. It provides contexts for making decisions throughout the product's life, and gives a common direction for all stakeholders in software development. A product strategy describes the vision of what a project team tries to accomplish [Ebert and De Man, 2005]. Ambiguous

product vision and strategy will lead to difficulties in planning and scheduling development activities.

In some cases, the key stakeholders of a software project are not involved in time, and some “untechnical” stakeholders have poor understanding of an application. Communicating between end-users and software developers is difficult because they must find a common language to exchange information. Developing enough shared vocabulary for communication can often take a while in a project which means that developers need to guess what the end-users require at the start of requirements part.

Dependencies are the relationships among tasks which determine the order in which activities need to be performed and project dependencies establish the links, and the types of links, between all the tasks of a project. One task may have multiple preceding tasks and succeeding tasks. Project dependencies also include dependencies between projects. Typical project dependencies are deliverables from other projects that one project requires in order to reach completion. This ensures that there is no confusion going forward. Project dependencies are two-fold, they can be inbound or outbound. Inbound dependencies are relationships that several projects deliver production environment to one project while outbound dependencies represent one project deliver production environment to other projects for their progress [Shiv and Doraiswamy, 2012]. Without clarity on the inbound and outbound dependencies, it will create considerable delivery challenges and have a huge impact on the business benefits.

A business case contains the reasoning for initiating a project or task, it describes the benefits, costs and impact, plus a calculation of the financial case. It is, however, difficult to capture all the factors at the first stage of a project.

Many requirements specifications are incomplete in that they only specify what the system shall do under normal conditions and thereby fail to specify what the system shall do under exceptional conditions. This incompleteness forces the developers to guess stakeholders’ intentions or to let these requirements fall through the cracks completely. [Firesmith, 2005]

The root causes lead to uncertainties in software development. There are four early project symptoms showing the requirements uncertainty, as discussed below.

The conflict of interest is a set of circumstances that creates a risk that professional judgment or actions regarding a primary interest will be unduly influenced by a secondary interest [Ebert and De Man, 2005]. This conflict can appear between clients, users and modelers, which will drive to an unclear propriety requirements distribution.

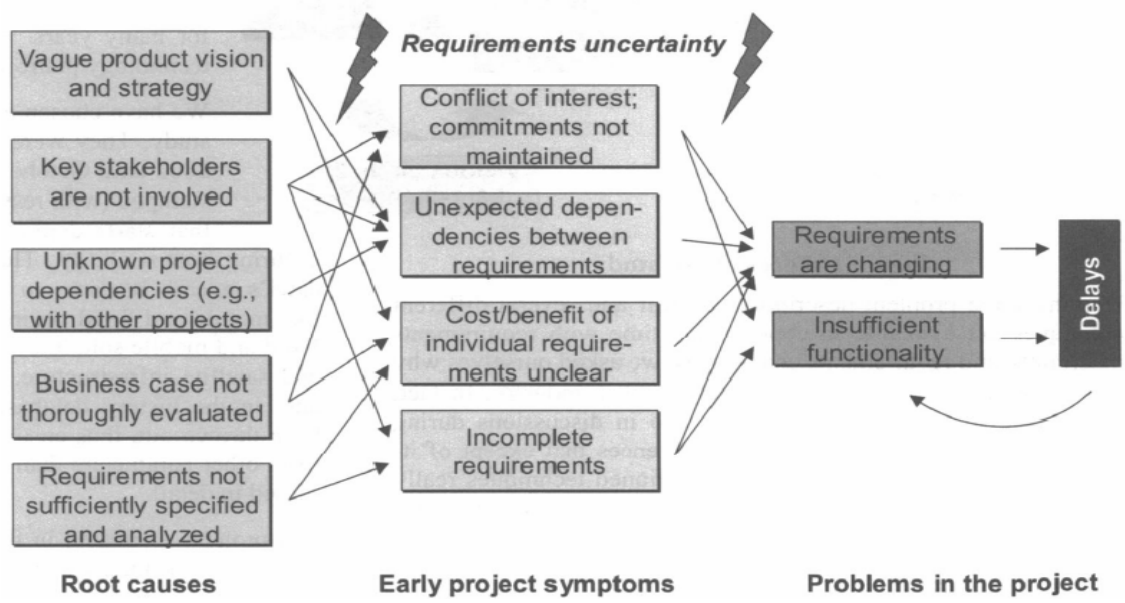

Figure 3. Causes of requirements uncertainty [Ebert and De Man, 2005]

Requirements dependency is the relationship between requirements and plays a role as the basis for change propagation analysis. Different dependency types can reflect the complex relationships between requirements at both structural and semantic levels [Zhang et al., 2013]. Unexpected dependencies between requirements increase the difficulties in understanding the requirements and influence many software engineering activities such as architecture design [Khan et al. 2008], product release planning [Carlshamre and Regnell, 2000] and change impact analysis.

If the analysis of business cases is not done sufficiently, and the vision and scope are not clearly specified, then the same easily happens to the product features and requirements. Consequently, it is very hard to accurately estimate the cost of product implementation at the requirements analysis stage, which will lead to an unclear decision on dealing with these requirements.

Incompleteness in requirements can be missing and vague information in requirements analysis models, individual requirement statements, and metadata describing individual requirements [Firesmith, 2005]. Requirements can be represented in graphical models, text and mathematical specifications. An incomplete requirements model fails in assigning proper value to every needed element of this model. For example, a use case model represents the behavior of a system by specifying systems external actors, the use cases, and relationships of actors and use cases [Firesmith, 2005]. It becomes incomplete if the value of a use case's attributes (e.g. the use case name, its goals, preconditions, etc.) is not properly specified. An individual requirement is incomplete if it fails to contain all necessary information to avoid ambiguity

[Firesmith, 2005]. For example， many functional requirements are incomplete because they only specify what the system will perform in normal conditions rather than in exceptional conditions, which will lead to developers guessing the stakeholders intentions or else they let these requirements fall through the cracks completely [Firesmith, 2005]. A requirement's metadata is data about associated project-unique identifier, priority, and status which can describe or characterize the requirements. Some typical types of requirements metadata may be found to be missing, such as categorization, criticality to customers and users, estimated cost range, frequency of execution, implementation status, etc. [Firesmith, 2005].

2.3.3. Representing requirements uncertainties in requirements models

Requirements uncertainty can be represented as annotations in a requirements document or in graphical models. In this thesis, we focus on how the uncertainties are identified and represented in models. Models are used to help with requirements elicitation, recording current understanding, communication, requirements development, and exploration of alternative high-level designs in the RE process [Salay et al., 2012]. During the process of creating models, it is common to uncover uncertainty over the contents and structure of the model. Since the modeling process often implicitly involves identifying model uncertainties and then resolving them through further elicitation or decision making, it is necessary to express uncertainty in RE models explicitly, and to capture such uncertainty for reducing the risks in making decisions and eliciting information as part of the modeling process.

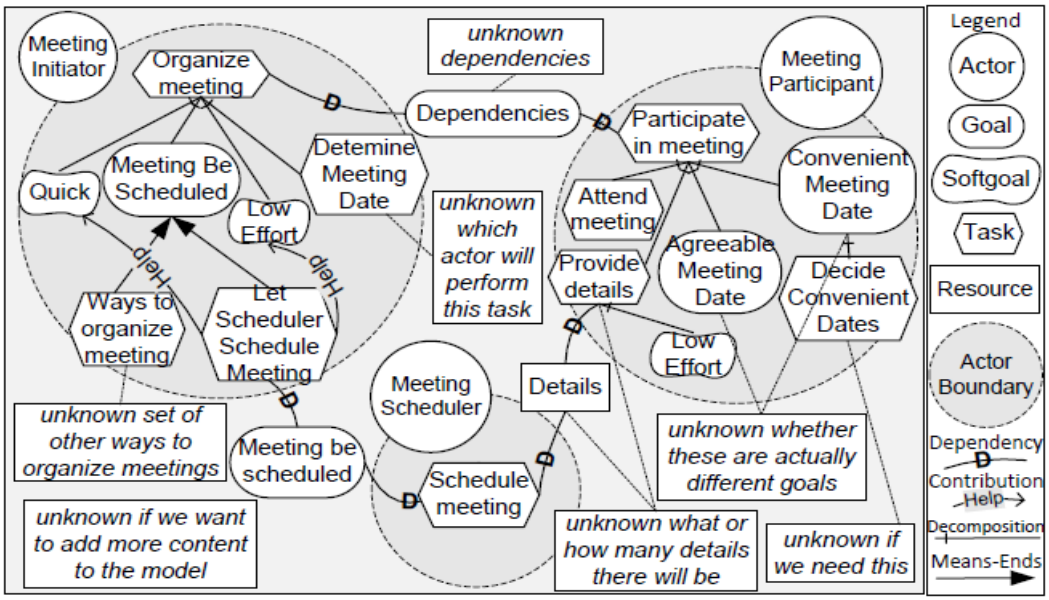

Figure 4. An early RE diagram with annotated uncertainty for the meeting scheduler example. [Salay et al., 2012]

A computer-based meeting scheduler application [Salay et al., 2012] is used as an example to illustrate how uncertainty is presented in a requirement model. The application supports the setting up of meetings in a company. For each meeting request, the *meeting scheduler* will help to *determine a meeting date* and location for effective participation. The system would fix a meeting date and location that meets the *meeting participants*' need. All potential participants' information will be collected by the *meeting initiator* about their availability to meet during a date range includes the exclusion dates and preference dates. Then, the *meeting scheduler* comes up with a proposed date. The date must not be one of the exclusion dates, and should ideally belong to as many preference sets as possible. *Participants* would *agree to* a meeting date once an acceptable date has been found [Yu, 1997].

Figure 4 shows the process of scheduling a meeting by using i* modeling framework [Salay et al., 2012]. The i* model focuses on the early-phase of requirements engineering activities, such as how the intended system would meet organizational goals, why the system is needed, what alternatives might exist, what the implications of the alternatives are for various stakeholders, and how the stakeholders' interests and concerns might be addressed [Yu, 1997], while most other RE modeling techniques stress on later phase on the completeness, consistency and automated verification of requirements [Yu, 1997].

The central component of i* framework is the actor, and one actor will achieve its goals by depending on other actors. In Figure 4, there are three actors which are represented in circles, i.e. meeting initiator, meeting participant and meeting scheduler. Intentional elements, such as goals, tasks, resources and softgoals, appear not only as external dependencies, but also be linked by means-ends relationships and task-decompositions. On the meeting initiator side, the main task is to *organize meeting.* This task shall fulfill several goals, and include several subtasks. Softgoals are used to filter the subtasks. For example, *quick* and *low effort* scheduler ways will be chosen for the meeting organization. *Meeting be scheduled* is the goal of the main task, and it will be achieved by two subtasks. One is the *let scheduler schedule meeting* and the other one is about the *ways to organize meeting*. The former one has a goal-dependency with the actor of *meeting scheduler*, while an uncertainty appears with the latter task. It is about the unknown set of other ways to organize meetings and shown in a rectangular in the lower left corner in Figure 4. The subtask of organize a meeting is *determining meeting date*. But there is not a particular description on the actor who will perform this task, which forms an uncertainty in the process. Organizing a meeting has an unknown goal-dependency with the main task performed by the actor *meeting participant - participating in meeting*, which need a further check about what exactly this

dependency is. Participating in a meeting also can include hierarchically with two subtasks and two goals. One task is to *attend meetings*, and another one is to *provide details* for meeting schedulers. The vague specification of the meeting details will impact on further meeting scheduling. Besides, the two goals of participating meetings are picking up a *convenient dates* and *agreeable meeting date*, which are probably mixed concepts if there is no further explanation. A subtask of picking up a convenient meeting date is to *decide convenient dates*, which may be unnecessary and deleted later. At last, there is an unknown when we want to add more content into this model, which is shown in the rectangular in the lower right corner in Figure 4.

According to Salay et al. [2012], all these uncertainties can be divided into three classes. The first kind of uncertainty is caused by the gap of domain knowledge among the modelers. Domain knowledge is knowledge about the operation of target system. Each stakeholder has his/her own education background and may result in different perceptions of the domain [Kotonya and Sommerville, 1998]. It is also the reason why communication between end-users and software developers is difficult. This will lead to one of the early project symptoms, which is about the benefit of individual requirements unclear. Therefore, all stakeholders need to find a common language for communication. Developing enough shared vocabulary to communicate can often take a while and that is the reason why uncertainty happens. For example, the modelers may quickly have questions like what are the other options of meetings organization quickly. The second type of uncertainty is caused by stakeholders' disagreements, such as whether the meeting initiator or participants has the responsibility of picking up the meeting dates? This kind of uncertainty is related with the conflict of interests between stakeholders. The last kind of uncertainty is the continuous change of model details and the modelers' attitude of adding details later. It brings the incompleteness requirements into the development process.

In summary, incompleteness is treated as a symptom of uncertainty. Looking back to Figure 2, we can see that the uncertainty in requirements will lead to changing requirements and insufficient functionality, and the final result will be a delay in the whole software development process and it will lead to the software "instability", [Ebert and De Man, 2005] which results in many consequential problems such as late delivery, over-run budget, poor quality, etc. In a word, the most important thing at the requirements engineering stage is how to extract the right requirements and how to deal with the changing requirements [Ebert and De Man, 2005]. This thesis intends to find a way to detect incompleteness in requirements and to elicit the missing requirements using the metamodeling technique.

3. Model, metamodel and MetaEdit+

After specifying requirements, models are constructed to represent the software system from different perspectives. For example, an ER diagram specifies the entities of a software system and their static structure; a use case diagram specifies the interaction between users and the system, as well as the behaviors the system possesses. Each model has its own metamodel which specifies the basic elements, relationships and attributes for the model. In this chapter, the concepts of a model and its metamodel are presented, together with an introduction of the metamodeling tool MetaEdit+.

3.1. Concept of model and metamodel

A model is a simplified representation or an abstraction of a certain reality [Bézivin, 2005]. A model usually focuses on one particular aspect to specify a problem domain and to represent the design of a system, such as the static data and their structure, the expected behavior of a system and its interactions with users, the possible functions a system provides, etc. Therefore, a system is usually described by many different models, and each model captures a specific aspect.

Modeling in software engineering is the process of creating a model for a system by applying formal modeling techniques, and this process allows one to deal with the world in an easier manner, avoiding the complexity, danger and irreversibility of reality [Rothenberg, 1989].

A metamodel is a conceptual model of a modelling technique [Brinkkemper, 1990]. It captures essential properties and features of a model, and includes syntax and semantics about how the models and programs mean and behave in both textual and graphical way [Sammut and Willans, 2008]. As shown in Figure 5, a metamodel provides explanation and definition of relationships between the different components of the applied model itself. A model is an abstraction of phenomena in the real world while a metamodel is another abstraction, highlighting properties of the related model [metamodel.net, 2014]. A model conforms to its metamodel in the same way that a computer program conforms to the grammar of the programming language in which it is written. In summary, a metamodel can be seen as a model's model.

Metamodelling is the process of the conceptualization of a modelling technique [Brinkkemper, 1990]. It is the construction of a collection of concepts (e.g. things, terms, etc.) within a certain domain. It typically involves studying the output and input relationships and then fitting right metamodels to represent that behavior. Metamodeling is also defined as a process of generating the specific metamodels, which

is the analysis, construction and development of the frames, rules, constraints, models and theories applicable and useful for modeling a predefined class of problems [metamodel.net, 2014]. A metamodeling language is used to specify the abstract syntax of models.

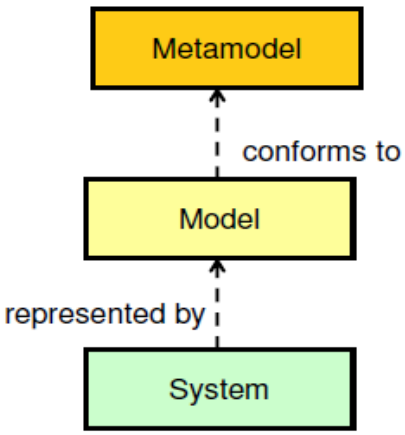

Figure 5. Relationship between metamodel, model and system [Génova, 2009].

3.2. **GOPRR metamodeling language**

MetaEdit+ is used as the metamodeling tool in this thesis work. It provides solutions to the detection and analysis of requirements incompleteness, with its specific generators. GOPRR, as the metamodeling language which is used by MetaEdit+, stands for five main elements used in the metamodeling process, and they are Graph, Object, Property, Role and Relationship [Tolvanen et al., 2007].

The O in GOPRR represents objects, which are the main parts of the model design. They are elements that connect together and are often reused. Objects are represented as rectangles in the metamodel. Figure 6 shows a graphical representation of the metamodel of an Entity-Relationship (ER) Model. In the ER metamodel, there are three main objects, i.e. *Entity*, *Relationship* and *Attribute*. The *AbstractER* is an abstraction object while the *Note* is the object to add additional text anywhere in the graphs.

The first R in GOPRR represents a Relationship, which defines objects' connections. In other words, a relationship is the basis to form a binding between objects. It is represented as a diamond in the metamodeling. In Figure 6, there are three relationships, and the name of them are *in relationship*, *Supertype-subtype* and *Attribute of*.

The second R in GOPRR represents Roles, which links an object with a relationship in a metamodel. It specifies how the connected object joins the relationship with another

object(s). Therefore, in a binary relationship, there are always two roles connecting the relationship with two joined objects. The Roles are shown as circles in the metamodel. For example, in Figure 6, the roles in the relationship *Supertype-subtype* are generalization and specialization.

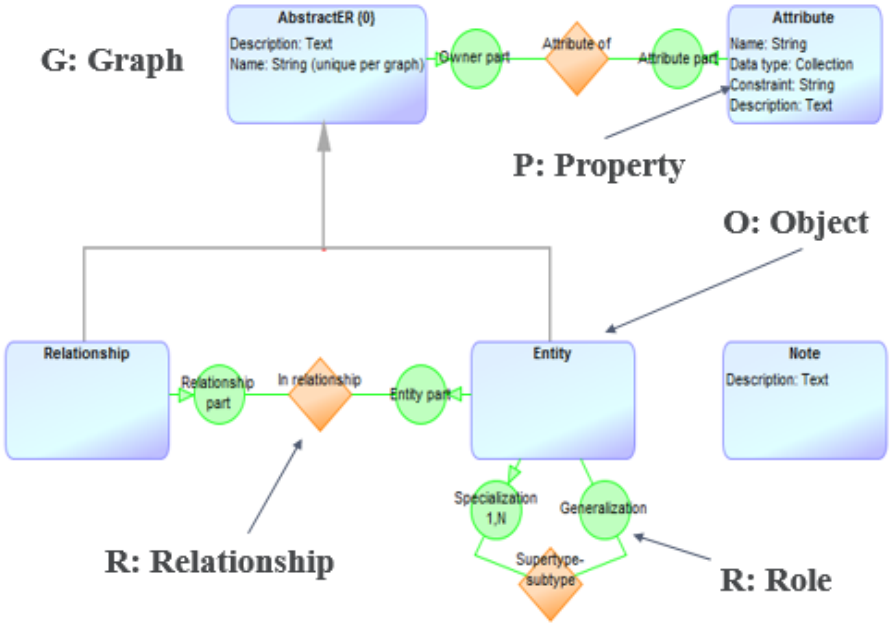

Figure 6. The original graphical representation of the metamodel of ER diagram in MetaEdit+

The G in GOPRR represents the Graph, as the graph shown in Figure 6. It consists of instances of all other types of elements to form a specification of one modeling language. A separate metamodel can be made with all details of each language. Integration between languages can be specified in a metamodel for several graph types.

At last, the P in GOPRR represents Property. Logically, a property defines the attributes of the instances of the other four elements in the GOPRR metamodeling language, i.e. instances of an object type, relationship type, role types, and the graph. A property can be set up with different data types, such as string, text, number, Boolean, etc. For example, the properties of the ER metamodel graph will be the graph's *Name*, *Status* and *Description*. *Name* and *Status* are in the string type while the *Description* is the text type property,

3.3. **MetaEdit+**

CASE (computer-aided software engineering) is the use of a computer-assisted method to organize and control the development of software, especially in large, complex

projects involving many software components and people. As a CASE tool, MetaEdit+ can help in analyzing, designing and implementing in the process of software development [Kelly and Lyytine, 2005].

The architecture of MetaEdit+ is given in Figure 7 [Kelly and Lyytinen, 2005]. The core of this environment is the *MetaEngine*, which operates the conceptual data through a well-defined service protocol [Smolander and Kari, 1993], or in other words, handles the implementation of conceptual data modeling. The software uses the *MetaEngine* to access and operate the data from the repository, and after the obtaining process, the data of model or metamodel can be operated by different tools in MetaEdit+ [Kelly and Lyytinen, 2005].

Model editing tools, i.e. the diagram editor, the matrix editor, and the table editor, are used to create, modify and delete model instances or their parts. In addition, these tools can be used to view the model instances from different representational viewpoints, which can derive new information from existing design information as well.

Browsers are used for retrieving design objects and their instances from the repository for reuse and review. They are also used for linking design objects for "traceability" and "memorization" [Kelly and Lyytinen, 2005], annotating model instances, finding specific locations in the design space, or maintaining conversations about design issues.

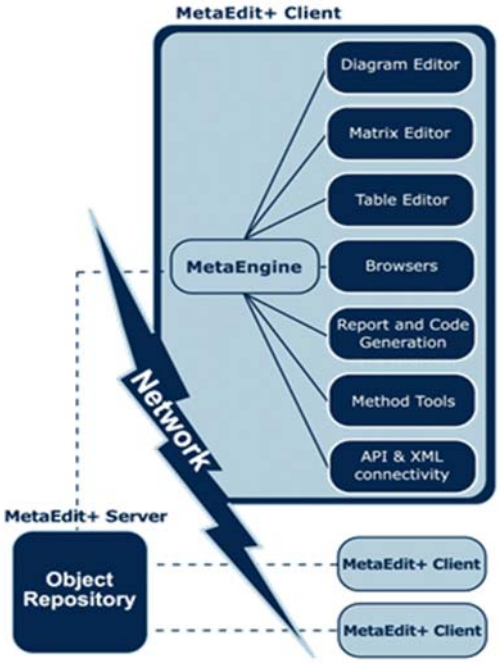

Figure 7. MetaEdit+ Architecture [Kelly and Lyytinen, 2005]

MetaEdit+ provides a generator and report function, i.e. the report and code

generation, to show expected results in a report format. The Generator Editor is an interactive development environment for creating, editing and managing generators. It allows users to view, edit and run available generators, and to create new generators for specific needs. After the detection of generator, the required information will be shown in the report and the report has its own editor for editing the details of report.

Method management tools are used for method specification, management and retrieval, and also the tools to construct the individual elements into an integrated method (model). MetaEdit+ Workbench contains specific tools for creating and maintaining each of the GOPPRR elements, i.e. Graph Tool, Object Tool, Relationship Tool, Role Tool and Property Tool. As shown in Figure 8, by using the Graph Tool, we can modify the objects, relationships and roles by their own tools. For example, the *Object [GOPRR]* in the objects bar can be edited by the Object Tool with its name, ancestor, properties and descriptions, which is shown as the top right corner in Figure 8. The Property Tool allows developers to edit the property's name, widget, datatype, default value, value regex and description. In a similar way, Relationship Tool and Role Tool can do the modification to their related elements. Besides, MetaEdit+ provides Symbol Editor (in the red rectangle in Figure 8) and Icon Editor (in the blue rectangle in Figure 8) to edit the elements symbol in the metamodeling process and show them with icon in the elements bar.

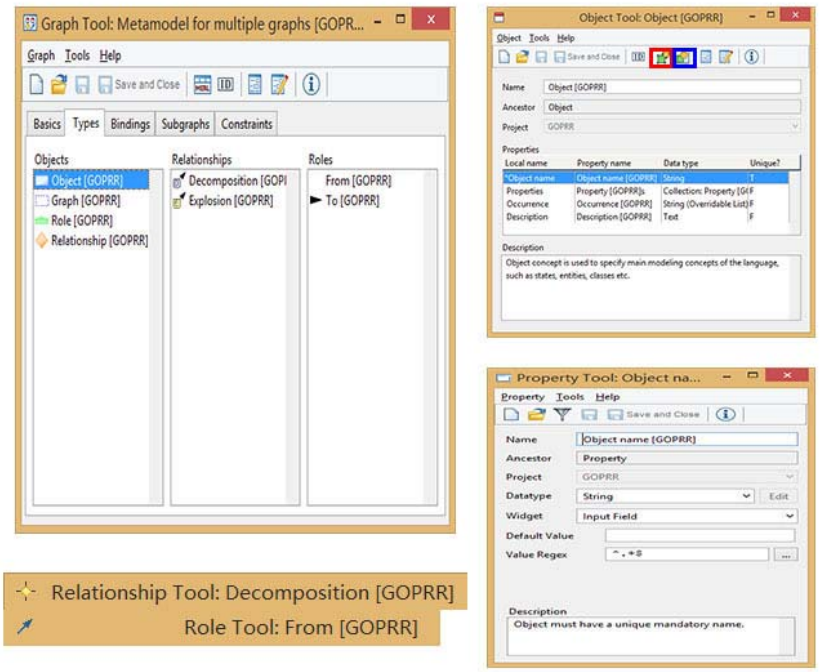

Figure 8. Method management tools used in MetaEdit+

MetaEdit+ versions without the API provide only a proprietary binary format for import and export. The API product adds support for import and export in XML format. The main function of this format is to enable exchange of data between MetaEdit+ and other programs.

4. Mapping business rules onto software design models

By using metamodeling specification, we can create models with complete elements and correct structure. Other model information can be obtained from the different types of requirements. With the increasing interest on business rules, there is a trend to create a dedicated rule-centric modeling framework and methodology to link business rules to realistic software model design [Wan-Kadir and Loucopoulos, 2003].

4.1. MBRM framework

A Manchester Business Rules Management (MBRM) approach has been introduced to relate the given business rules to the specification of software designs elements [Wan-Kadir and Loucopoulos, 2003]. The MBRM framework has two main components, i.e. the MBPM mapping metamodel and its related architectural components, as shown in Figure 9.

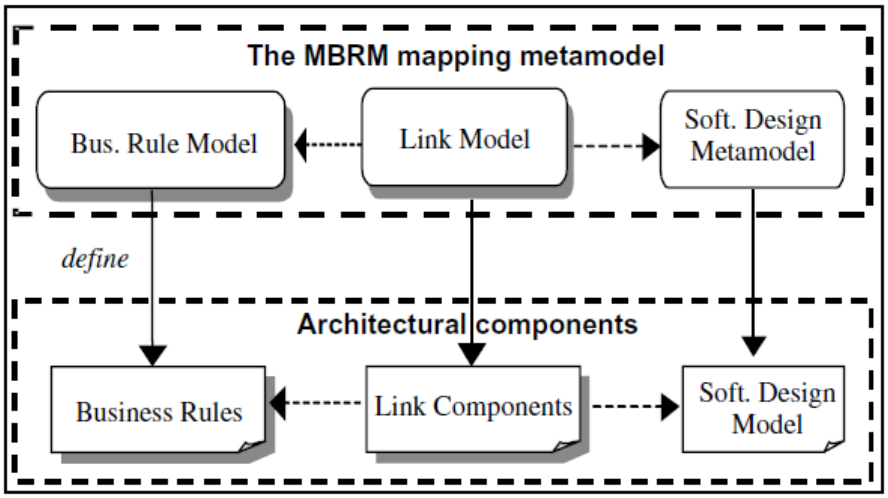

Figure 9. The structure of the MBRM mapping [Wan-Kadir and Loucopoulos, 2003]

On the metamodel level, the business rule model captures and specifies business rules structures and semantics, while the relationship between business rules and metamodel of software design elements is defined by a link model [Wan-Kadir and Loucopoulos, 2003]. Based on the business rules taxonomy introduced in Figure 2, this business rules model classified them in different layers. As shown in Figure 10, the business rules are grouped into three main types, i.e. constrains, actions assertion and derivation.

A constraint rule checks the result of execution of business event on a subject. A subject can be a term, which can be a word or a phrase that is related to the business, or a fact. One constraint can be divided into a mandatory constraint or a guideline. A

business event will be denied when it violates the mandatory constraint and there will be a warning to users if it violates the guideline rules [Yu, 1997].

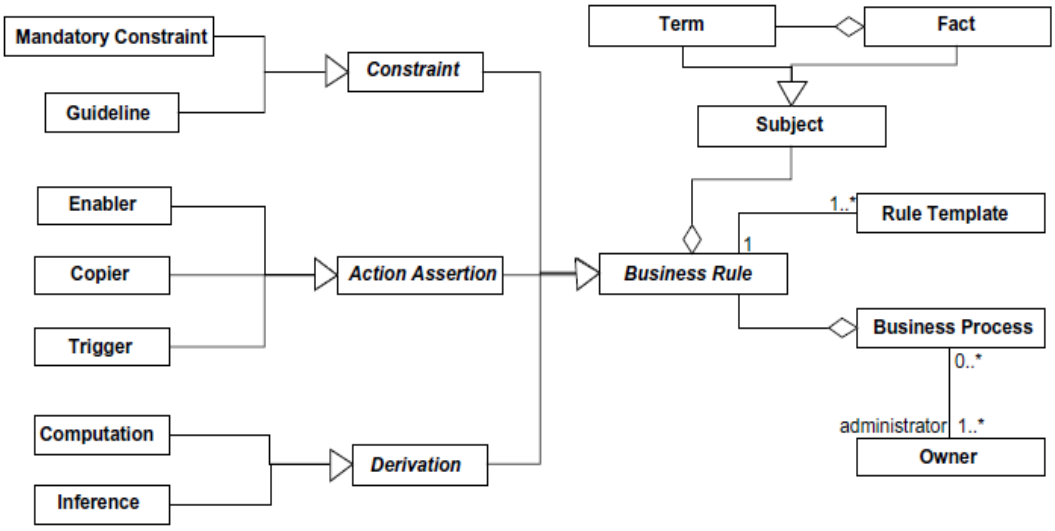

Figure 10. Business rules metamodel [Wan-Kadir and Loucopoulos, 2003].

An action assertion specifies the action that should be activated on the occurrence of a certain event or on satisfaction of a certain condition. It can be divided into three different types i.e. enablers, copiers, and triggers. An enabler rule enables or disables a rule, operation, process, or procedure according to certain conditions. It also creates and deletes data under specified conditions. Copier is concerned with the use of existing data or value, for example, using a certain value to set the initial value of an object's attributes or to determine the way on how to present existing data. When a given condition is true or on the occurrence of a certain event, the trigger will cause operation, process, procedure, or rule to be executed [Yu, 1997].

A derivation generates a new fact based on the existing terms and facts. It can be divided into two types. Computation derives a new value by using a mathematical calculation while inference uses logical deduction or induction [Wan-Kadir and Loucopoulos, 2003] to derive a new fact. Inference rule is also used to represent the permission such as the user policy for data security. The definition of business rules can be one or more rule templates for later use and other business process.

In the conceptual modeling level, a business rule metamodel is created to separate syntax and semantics for modeling business rules and to increase the understanding and maintainability of business rules specification [Wan-Kadir and Loucopoulos, 2008]. It provides an excellent classification from the functional aspect of business rules and helps to revise the raw business rules [Wan-Kadir and Loucopoulos, 2003].

A link model is specified to link business rules with the design elements. As can be seen in Figure 11, every business rule is associated with the *RuleImplemention* in the Link model. The *RuleLink* class is inherited from the RuleImplemention. *DesingElementID* is an attribute of the *RuleLink* class, which shows which object is linked by the specific business rule. Both *isAutomated* and *isAutoChanged* are Boolean values. The former one returns the true value if the rule is implemented in the design element whilst the latter returns the true value if the software design can be automatically changed according to the changes of business rule using the supplied methods [Wan-Kadir and Loucopoulos, 2003]. The *RuleLink* class indicates the software implementation of business rules.

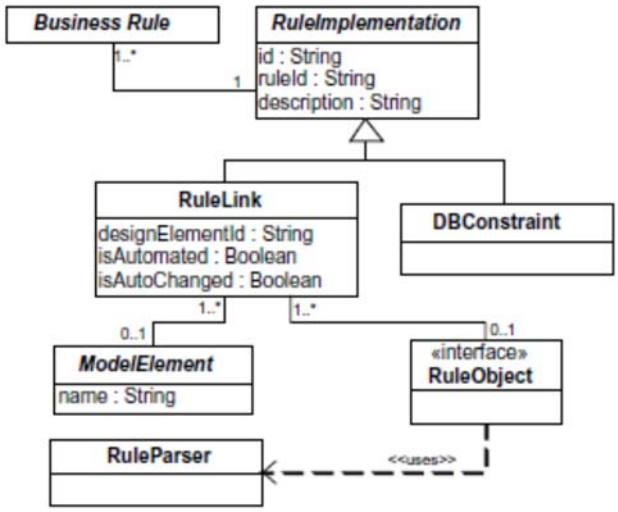

Figure 11. The Link model [Wan-Kadir and Loucopoulos, 2003]

Besides, the MBRM framework presents the traceability link between business rules and their implementation in the design model. The traceability link is defined on the metamodel level, and connects the constructs specified in both the metamodel of business rules and the one of software design models. Different types of business rules are linked to the different design elements. Figure 12 shows an example of the link model on the architectural component level. There are two business rules to describe the process of patient registration and the linked object is the class Patient. The first one is a constraint, i.e. *a patient must have registration number*. The Rule link object is defined as the rlOBJ1 in this diagram. It has identified attributes such as *id, description* and *ruled*, and the related attribute of *designElementid* is *Patient()*, which means this attribute will show the information of patient's registration number. The second business rule provides a calculation of the registration number, i.e. *a patient registration*

number is computed as the largest patient registration number + 1, which is defined by the attribute of *getRegNo()*.

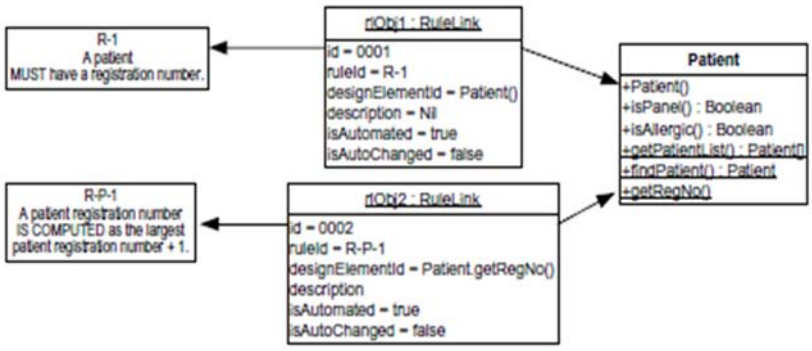

Figure 12. Part of examples of rule link objects [Wan-Kadir and Loucopoulos, 2003].

4.2. **A Link Model connecting business rules with revised ER model**

In a similar way, when replacing the class diagram with another software design model, e.g. Entity Relationship model, the links connect the business rules with different ER model elements. Accordingly, business rules describing associations or relationships between business terms shall be put into force when defining entities, relationships and attributes for a project; while other business rules putting constraints or conditions onto a software system behavior might lack enough connection with the elements in an ER model.

An ER model specifies a static structure of a software system by representing entities and their relationships. The elements in its metamodel include entities, relationships, and attributes labelling entities and relationships. In addition to the basic ER model, some extensions, which are based on the Enhanced Entity Relationship (EER) model [Fidalgo et al., 2012], are taken into account in the thesis. These extensions reflex more precise concepts and their properties, and include concepts of the weakness of entities, supertype and subtype entities (along with the concept of specialization and generalization relationships), and the concept of aggregation. These are represented in Figure 13, i.e. a graphical representation of the extended ER metamodel using the GOPRR metamodeling language.

As shown in Figure 13, in addition to the metamodel presented in Figure 6, we include extensions such as the Aggregation relationship, *isWeak* property on the Entity and *isOverlapping* property on the Supertype-subtype relationship. The *isWeak* property is a Boolean value, which means if the value is true, the entity would be a weak entity in the model. In a similar way, the *isOverlapping* is also a Boolean property, which specifies an overlapping relationship between supertype and subtype entities when this value is true. At last, we add the *Aggregation* relationship between entities, with the role of low-level entities collection and its high-level entity. Besides, some

basic properties are already in the basic metamodel. Each role between a relationship and an entity has the *Cardinality* property, which records the values of connective between entities, such as one-to-one, one-to-many, many-to-many. One of the property in attribute is Constraint, which specifies the information of unique value and primary key of an attribute.

As introduced in the prior section, there are three categories of business rules. In the following paragraphs, we further study how the different categories of business rules are implies in an ER model.

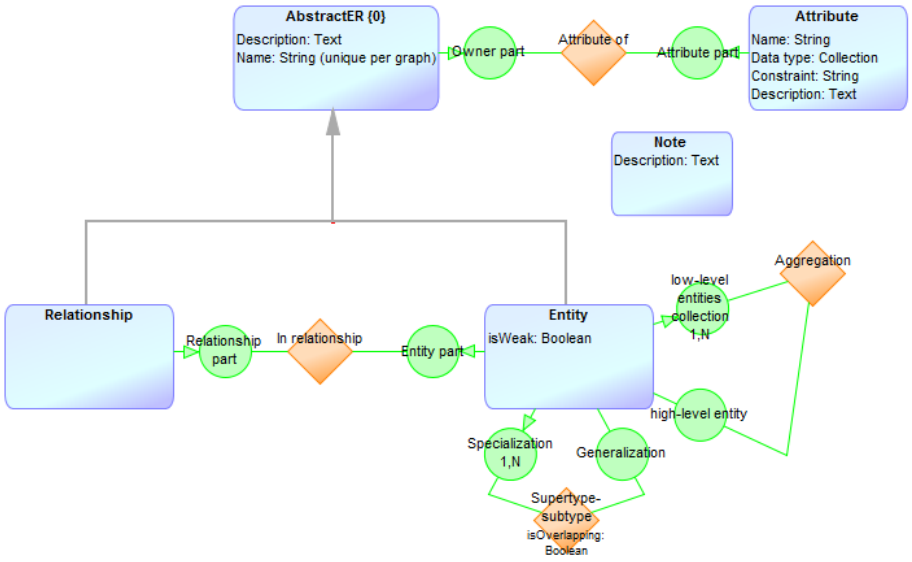

Figure 13. The revised ER metamodel

A *constraint* rule could consists of several facts, which is a statement that asserts a relationship between entities in a model. The information of relationship includes the cardinality in a relationship, type of relationships, such as generalization, specialization, or aggregation.

A *derivation* provides a piece of knowledge in an "if-then" clause which is related with some basic entities and their own attributes in a model. For example, the weakness of entities can be checked in this part.

An *action assertion* specifies the actions which should be activated on the occurrence of a certain event or on satisfaction of a certain condition. However, these kinds of business rules are more useful in a State Transform diagram, rather than in the ER diagram.

Business rules	Related ER diagram elements
Constraint	property of subtype and supertype (overlapping) entities, cardinality in a both relationship part and entity part, aggregation
Derivation	weakness of an entity, constraints of an attribute (primary key, unique values)
Action Assertion	-

Table 1. Link between business rules and ER diagram elements

Table 1 shows the major link between business rules and ER models, and this link model can simplify the check process. When a metamodel construct is not property specified in the concrete instance model, it is natural to note that business rules and their elaborated requirements might be missing, which leads to difficulties in specifying the model elements and requires a further check.

5. Incompleteness in an ER model

The business rule model defines and refines business rules and these business rule elements are implied in the software design models. The traceability links are represented in the link model in the MBRM framework. Vice versa, it is possible that any missing information in a design model can be traced back to the unclear/incomplete specification of business rules. Some specific metamodeling techniques can be used to detect the missing information in design model, and these incompleteness identified by modelers can imply the missing information about requirements (business rules), and notify the development team to elicit the omissions in requirements [Zhang et al., 2014].

In order to analyze the incomplete requirements, a list of questions are generated [Zhang et al., 2014] [Thanish et al., 2013] to be used as examples based on the EER model [Fidalgo et al., 2012], and in my thesis, all the incompleteness will represented with the ER model based on the revised ER metamodel. These incompleteness problems are unclear number of entities, weakness of an entity in the model, multiplicity of attribute value, type of attribute status (identifier or a part of a composite identifier), unclear relationship's degree, ambiguous relationship's connectivity, optionality of the occurrence of an entity in a relationship, unsure type of subtype (overlapping or disjoint), unsure multiple relationships (inclusive or exclusive), aggregation hierarchy redundancy, redundancy of relationship and unclear attribute belonging (to a relationship or an entity). Meanwhile, representation of these types of incompleteness in the modeling process is also provided in [Thanish et al., 2013].

5.1. Infrastructure incompleteness

All the questions can be divided into two aspects of EER diagram, i.e. structure and infrastructure. The structure comprises the shapes and lines in the ER diagram, whereas the infrastructure comprises the information concerning the various constructs which make up the structure [Thanish et al., 2013]. In terms of the EER metamodel which is described before, the structural items in an EER diagram correspond to the objects, whereas the infrastructure items in an EER diagram correspond to their properties.

For each category of incompleteness, we provide a general scenario and a specific example in order to explain how the specific kind of incompleteness can arise in practice, how it can be incorporated into the EER diagram convention.

5.1.1. Unclear number of entities

Sometimes, it is difficult to decide whether an object should be represented as one entity or two entities in a relationship [Thanish et al., 2003]. For example, in a company, if a person has more than one employment, then such a person may have more than one employee ID. Therefore, it may be necessary to create more than one entity to describe the construction, which is shown as Figure 14. The dashed-line entity may appear if a person can be more than one employment.

Figure 14. One or two entity [Thanish et al., 2013]

5.1.2. Weakness of an entity in the model

A weak entity cannot be uniquely identified by its attributes along [Connolly and Begg, 2009]. For example, in a company, we have the entity of employee and their dependents (like their parents). When a person leaves the company, both the employee and dependent entities (maybe with other entities) will be removed. Therefore, the dependent could be a weak entity and a bigger dashed outline will be added as a symbol, which is shown in Figure 15.

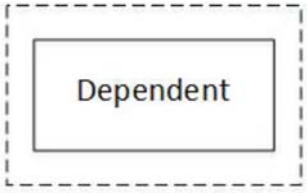

Figure 15. Weak entity Dependent [Thanish et al., 2013]

5.1.3. Multiplicity of attribute value

This incompleteness focus on whether an attribute is a multivalued attribute or not. Suppose that a company would like details of all degrees held by employees. However, it seems that the only information available might be the highest degree. If it is possible to obtain all degrees then the attribute is multi-valued. Otherwise, the attribute is single-valued.

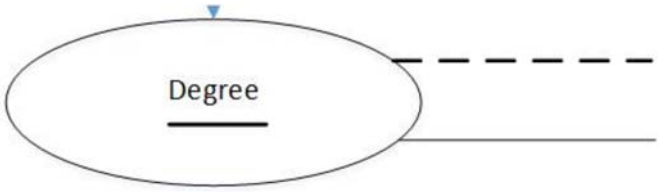

Figure 16. Table of staff hobbies in a company [Thanish et al., 2013]

In this case, the potentially multi-valued attribute is modeled with an underline and linked with an extra dashed line.

5.1.4. Type of attribute status

A particular attribute is known to be an identifier, but sometimes it may be impossible to determine whether it is actually a part of a composite identifier.

For example, two persons' social security numbers the same if they come from different countries. Therefore, the country should be one part of the composite identifier. In ER model, the concept of identifier is much similar with the concept of primary key in a physical database design, which is shown as Figure 17. The other primary key's underline and outline are both dashed.

Figure 17. Identifier or part of a Composite Identifier? [Thanish et al., 2003]

5.1.5. Unclear relationship's degree

Degree of a relationship is the number of participating entity types in a relationship [Connolly and Begg, 2009]. In other words, the degree indicates the number of entity types involved in a relationship. For example, a relationship of degree two is a binary relationship while the degree three is a ternary relationship. All the ternary relationships will be marked as the complex relationship with dashed outline as shown in Figure 18.

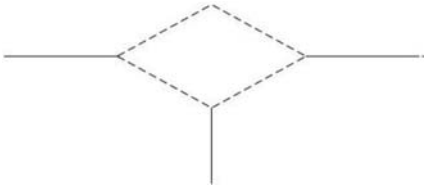

Figure 18. Degree of a relationship [Thanish et al., 2013]

5.1.6. Ambiguous relationship's connectivity

The connectivity of a relationship (also known as cardinality) is the number of occurrences in one entity which are associated (or linked) to the number of occurrences in another, such as one to one, one to many and many to many. Sometimes, the number of connectivity of two entities is not easy to be determined at the early stage of requirements engineering. For example, in a company, if there is no business rule that for each department there must exist an employee who manages the department then the connectivity will be the missing information shown in Figure 19.

Figure 19. Connectivity of a relationship [Thanish et al., 2003]

5.1.7. Optionality of the occurrence of an entity in a relationship

Sometimes it is not easy to determine whether an entity's occurrence in a relationship is mandatory or optional. In other word, it may be impossible to decide whether an entity will appear in a relationship. For example, it might be unclear whether there will be an employee to manage a department, and the optional occurrence entity is drawn with a circle which is shown as Figure 20.

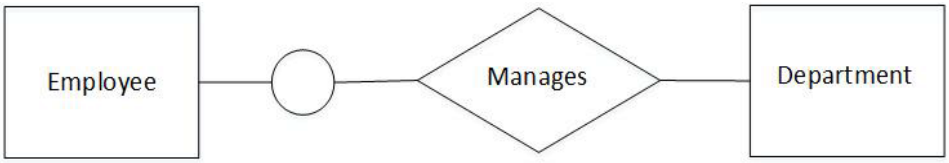

Figure 20. Optionality of the Occurrence of an Entity Instance in a Relationship [Thanish et al., 2003]

5.1.8. Unsure type of subtype

In a database, generalization is a "bottom-up approach" in which several lower level entities composite a higher level entity [Connolly and Begg, 2009]. For example, in a vehicle generalization hierarchy, the subtype entities of the supertype entity *Vehicle* could be *Trucks* and *Boats*. Analysis has not yet determined whether an amphibious vehicle is to be classifed as a truck or a boat or both or neither (i.e. a separate entity class). Thus is not known whether the subtypes are overlapping or disjoint [Thanish et al., 2003]. The mark o/d will be represented in the physical database design when this situation happens, which is shown as Figure 21.

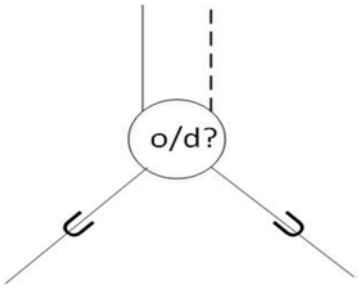

Figure 21. Subtypes overlapping or disjoint [Thanish et al., 2003]

5.1.9. **Unsure multiple relationships**

In an exclusive subtype relationship, each instance in the supertype can relate to one and only one subtype. For example, each employee in a company can get trained by a coach or is allocated to a mentor. However, there may be a business rule defining circumstances where an employee has both a coach and a mentor. This situation is shown as Figure 22.

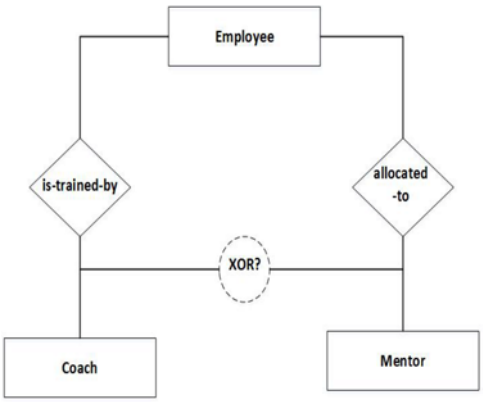

Figure 22. Exclusion Constraint: Exclusive or Inclusive [Thanish et al., 2003]

5.1.10. **Aggregation hierarchy redundancy**

Aggregation is a different supertype-subtype abstraction with generalization. Generalization shows an “is-a” relationship between subtype and supertype entities, while aggregation shows a “has-a” relationship. For example, in Figure 23, each product comprises a hardware component and a software component. However, with the business rules change, the hardware component and the software component may have to be sold separately as different products, which is shown as Figure 23. A potential change to the business affects whether it is appropriate to model some entity classes as being a part of an aggregation hierarchy [Thanish et al., 2003].

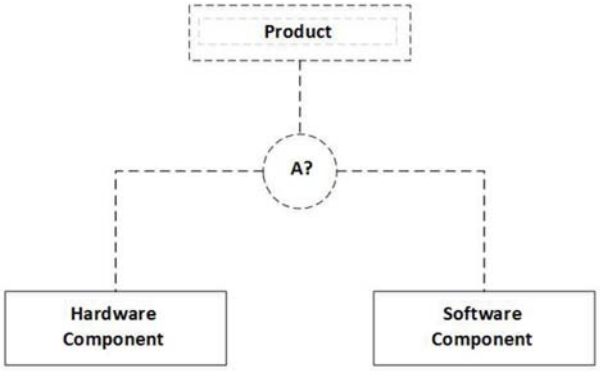

Figure 23. Completeness or Incompleteness of an Aggregation Hierarchy [Thanish et al., 2003]

5.2. Structural Incompleteness

In this section, there are two modeling scenarios provided for incompleteness at the structural level which can be associated with the conceptual modeling process.

5.2.1. Redundancy of relationship

Suppose there will be a relationship Manages_Emp from the Employee entity class to itself and the relationships Manages and Works_In between the Employee and Department entity classes. The Manages_Emp may be redundant when it has the same meaning as the Manages relationship from Employee to Department, which is shown in Figure 24. The redundant relationship is dashed in the modeling process.

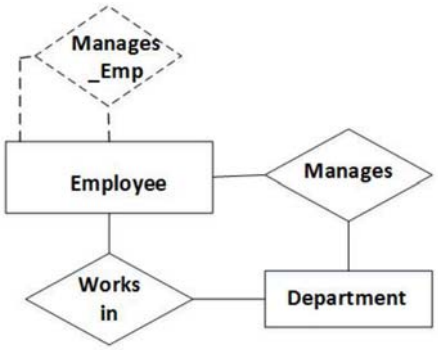

Figure 24. Relationship redundancy [Thanish et al., 2003]

5.2.2. Unclear attribute belonging

Sometimes, it is difficult to decide whether an attribute belongs to a relationship or an entity. For example, the date of an employee start to manage a department could be an attribute on either relationship Manages or entity Department, which is shown as Figure 25.

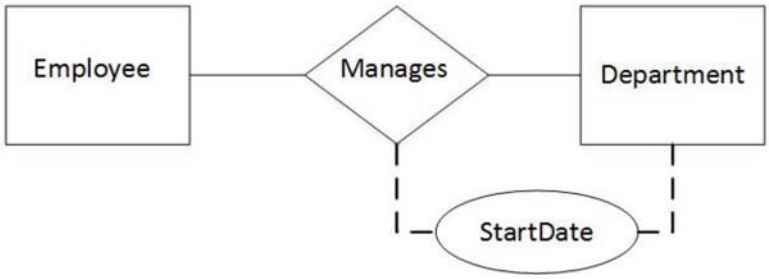

Figure 25. Attribute of a Relationship or an Entity [Thanish et al., 2003]

5.3. **Completed ER metamodel and link model**

According to the incompleteness list above, we add some necessary properties to the ER metamodel to make it capable for a more complete ER modeling process, such as the *isOptional* for optional entities, *isRedundant* for the unnecessary relationships, and the revised ER metamodel is shown as Figure 26.

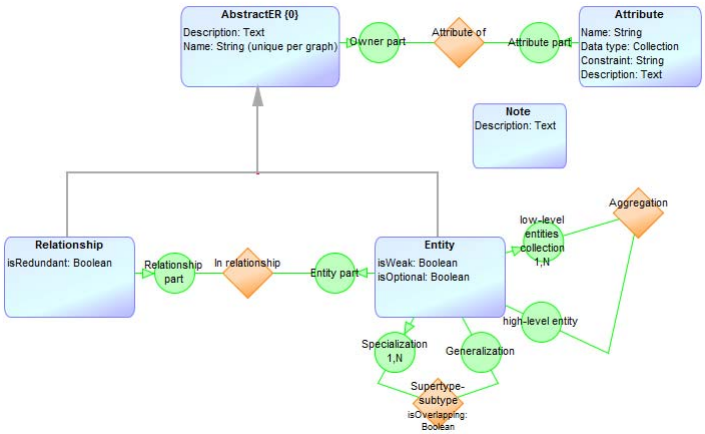

Figure 26. A revised ER metamodel

Moreover, based on the new metamodel, I link the incompleteness with the related element's property, and a revised link model is produced as shown in Table 2.

Business rule types	Category of Incompleteness in an ER model		Related property in metamodel
Constraint	5.1.5	relationship degree	String and number commands
	5.1.6	connective in relationship	Roles. Entity_part. Cardinality
	5.1.8	overlapping or disjoint hierarchy	Supertype-Subtype. isOverlapping
	5.1.9	exclusive or inclusive hierarchy	-
	5.1.10	aggregation hierarchy	Aggregation
	5.2.1	relationship redundancy	Relationship. isRedundant
Derivation	5.1.7 (5.1.1)	optional entity (number of entity)	Entity. isOptional
	5.1.2	weak entity	Entity. isWeak
	5.1.3	multi-valued attribute	ERAttribute. Constraint
	5.1.4	composite identifier	ERAttribute. Constraint
	5.2.2	attribute belonging	ERAttribute. Constraint
Action Assertion	-	-	-

Table 2. Revised link model

6. Detecting and reporting the incompleteness in an ER model using MetaEdit+

In order to demonstrate my approach of detecting the incomplete information using metamodel specifications, an excerpt of the ER model specifying the meeting scheduling application (as presented in Section 2.3.3) has been constructed in MetaEdit+, as shown in Figure 27.

In the model, there are two main entities, i.e. *Employee* and *Meeting*. An *Employee* works in this organization, and has attributes such as the *Name*, the *Nationality*, and the *Security Number*. In addition, an *Employee* supports his/her own *Dependents*. An *Employee* has three sets of sub-entities, i.e. the *Initiator*, the *Scheduler* and the *Participant*. An *Initiator* is responsible for organizing the meetings while a *Scheduler* decides the necessary Items of a meeting, such as the meeting room, time, and other details. *Participants* can *Propose* the date of a meeting and *Attend* in a meeting on a specific date. These three entities connect to the *Employee* through a supertype-subtype relationship. Each *Meeting* has items documenting the details such as meeting room, time, etc. and attributes such as the *Title*, and *MeetingID*.

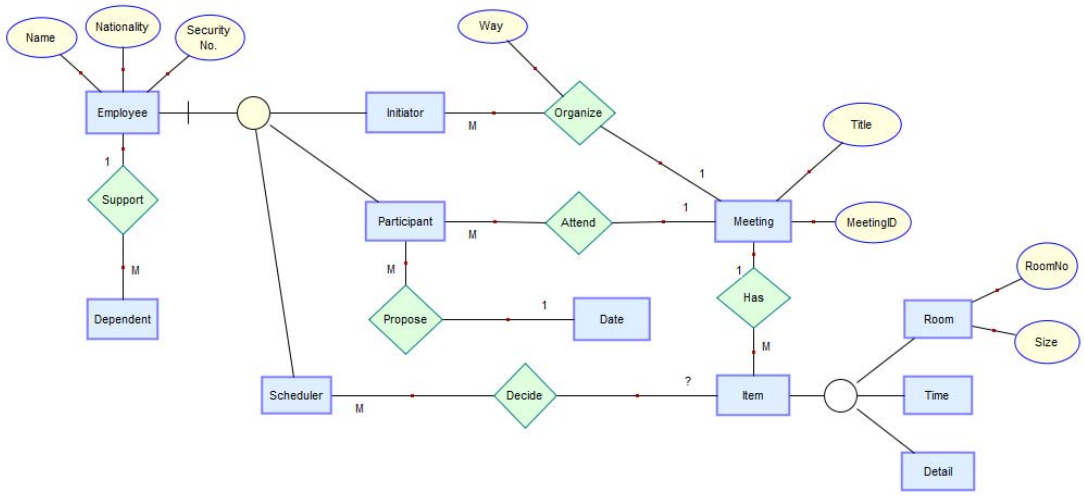

Figure 27. ER model of meeting scheduling process

Some unknowns discussed in section 2.3.3 have an impact on the ER models. They are given as following.

- The *unknown of what or how many details there will be* is a missing information in the entity *Item* and its related relationship *Decide*.

- The *unknown dependencies between different roles of employee* would be represented by the disjoint or overlapping relationship between the subtype entities of the *Employee*.
- The *unknown set of other ways to organize meetings* is the shown as an attribute of the relationship *organize.* It is unclear if this attribute has multiple values or not.

Known and unknown are two statuses of knowledge perceived and processed by individuals [Zhang et al., 2014]. The knowledge transfer in the RE process is started with the stakeholders, who possess a body of knowledge about the expected software or system, and requirements analysis transform that knowledge into requirements to the project team. The knowledge in the process can be distinguished between for states, and they are known-known (KK), known-unknown (KU), unknown-known (UK), unknown-unknown (UU). KKs refer to requirements which can be elicited from stakeholders clearly and explicitly. KUs refer to the knowledge which is realized by the requirements analyst but has not been able to elicit from the stakeholders. A missing business rule can be the simplest reason of a KU. UKs refer to knowledge that stakeholders has possessed, but requirements analyst is unaware. UUs refer to the knowledge which analyst is unaware of, and is not possessed by the stakeholders [Zhang et al., 2014]. In this case, for example, the unknown of how many details there will be is a missing information which can be marked as "1 or more". Since the modeler is aware of the unknown, it will be defined as a modeler's KU.

Except for the KUs which have been identified above, some other unclearness and missing information can be identified in the modeling process, as given below.

- The modeler is wondering if there is such a case that when an employee dies in the service of the company, the dependents continue to be supported by the company. As such a derivation rule is not explicitly written in the requirements document, the confusion about whether the entity *dependent* should be a weak entity or not exists.
- The modeler is wondering if the Participants are responsible for providing information of the *Date* in a meeting scheduling process. As such a derivation rule is not explicitly written in the requirements document, the confusion about whether the entity *Date* should be optional or not in the relationship exists.
- The modeler is wondering if the attribute *way* is an attribute of the relationship *organize* or be an attribute of other entities. As such a derivation

rule is not explicitly written in the requirements document, the confusion about the attribute belonging will exist.

- The modeler was wondering if two employees from different countries may have the same *Security Number*, which means that a composite primary key is necessary for the entity *Employee*. As such a derivation rule is not explicitly written in the requirements document, the confusion about whether the *Security Number* should be one part of the primary key of the entity *Employee* or not exists.
- The *Item* is an aggregation of three entities, i.e. *Detail*, *Room*, and *Time*. The modeler was wondering if there is need to un-bundle these three entities. As such a constraint rule is not explicitly written in, there may be confusion on the whether if there is redundancy in this aggregation relationship or not exists.

The completeness of an ER model can be analyzed based on its metamodel specification. Some of the above mentioned unknowns can be easily detected and reported. In MetaEdit+, the Generate and Edit Generator tool can be used for such a purpose. Based on the identified incompleteness in an ER model [Zhang et al., 2014] [Thanish et al., 2013], different generators can be created to report the possible missing information in an ER model.

All the incompleteness problems can be classified with the missing information in different ER elements and their related business rules, which is introduced in the link model before.

6.1. Incomplete property specification on relationships

Constraint can imply how an entity joins into a relationship with another entity. The implication is partly reflected in the properties of a relationship, such as the unclear cardinality of roles in a relationship, incomplete information in relationships between supertype/subtype entities, redundant relationship degree, and redundancy of relationships.

Since it is not easy to obtain all the connectivity information before modeling process, we can add one question mark or leave a blank to the proper place when the property value is unknown at the start of a project. As shown in Figure 27, the number of how many details can be provided by an employee is unknown, which means the cardinality of Provide relationship on Detail side is unknown.

Before modeling, we can use the Symbol Editor in the Role Tool to highlight the missing information in the ER model. As shown in Figure 28, we can add a judge condition for the Cardinality, and a red outline will be shown at the role *Entity part* when the value of Cardinality is a "?" mark while the *Entity part* will be filled with gray when the value of Cardinality is a Null.

The code that detects and sends the feedback about the unknown value of cardinality is shown in Figure 29.

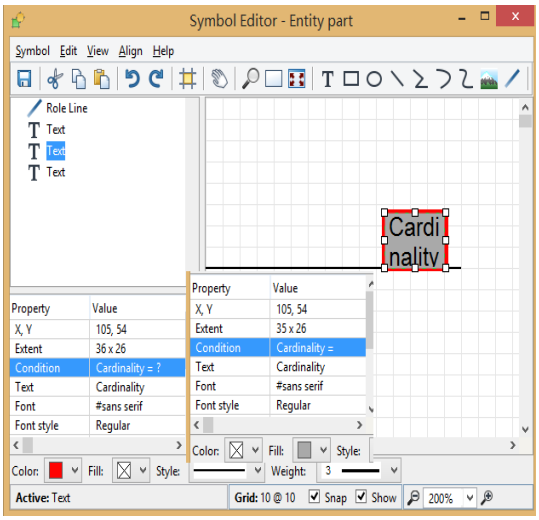

Figure 28. Symbol Editor of the Entity part

```
Report '_detectCardinality'
foreach .Relationship
{
    do >In relationship~Entity part
    {
     if :Cardinality = '?'
     then
        do  >In relationship~Relationship Role.Relationship
        {
        'The relationship ' id ' connects two entities '
        }

        do >In relationship.Entity {id ' and '}
     endif
     }


     do >In relationship~Entity part
     {
     if :Cardinality = '?'
     then

          do .Entity
          {
              ' Can only one or more than one ' id newline
          |' join the relationship?' newline 'Please clarify!'
          }
               endif
      }
  }
  endreport
```

Figure 29. Code of cardinality detection

Figure 30 shows the generated report. All the unknown cardinality and its relevant entity are highlighted in the report.

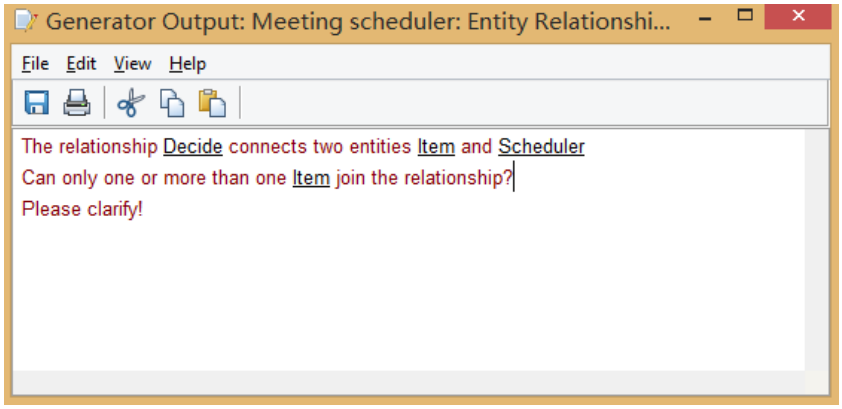

Figure 30. Result of Cardinality detection

Generalization and specialization are the basic concept in the original ER metamodel, but there is still a concept to constrain the relationships between the

supertype and subtype entities, i.e. disjoint or overlapping. A Boolean property about whether the subtype entities are overlapping or not is associated with the relationship Supertype-subtype. Here we can use the Symbol Editor to add the condition to detect the value of *isOverlapping*, as shown in Figure 31. When the value is true, an 'O' will be represented in the Subtype-supertype relationship whichs refer to an overlapping hierarchy; otherwise, a 'D' will be shown in the relationship as a symbol of a disjoint hierarchy.

At the same time, coding provides a detection to every subtype and supertype entity and capture their properties whether they are overlapping or disjoint. The code is shown as Figure 32.

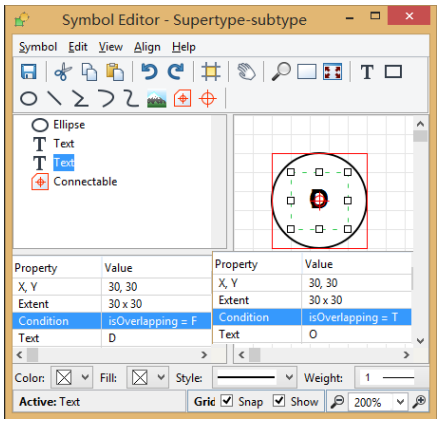

Figure 31. Symbol Editor of the Supertype-subtype relationship

```
Report '_overlappingSubEntities'

   foreach .Entity
   {
        do >Supertype\/subtype
         {
            if :isOverlapping =~ 'T'
            then
                  do ~Generalization part.Entity
                  {
                  'Can a ' id ' be either'
                  }

                  dowhile~Specialization.Entity
                  {
                     ' a ' id ' or'
                  }
                  ' but not all of them?' newline
                  'Please clarify!'
             endif
         }
   }
endreport
```

Figure 32. Code of overlapping or disjoint relationship detection

And the detecting result is given in Figure 33.

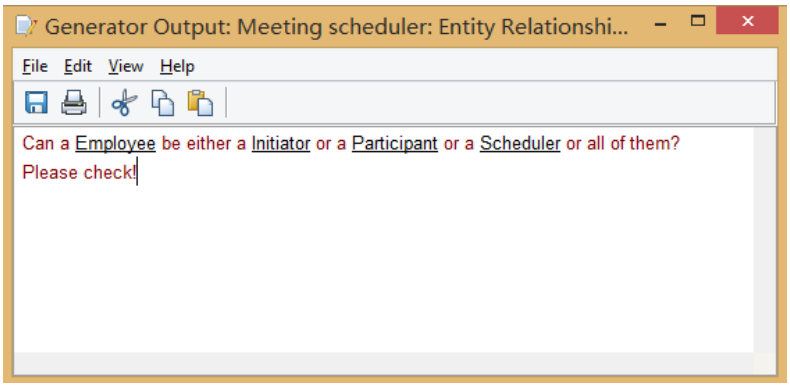

Figure 33. Overlapping or disjoint between subtype entities

Different from the overlapping and disjoint hierarchy, the inclusive or exclusive option is difficult to represent in the modeling process. Therefore, there is no solutions on the inclusive and exclusive detection by MetaEdit+.

Aggregation can be also detected by the generator and the result will be shown as Figure 34 to remind the modeler to check if there is a need to un-bundle the entities in the aggregation relationship.

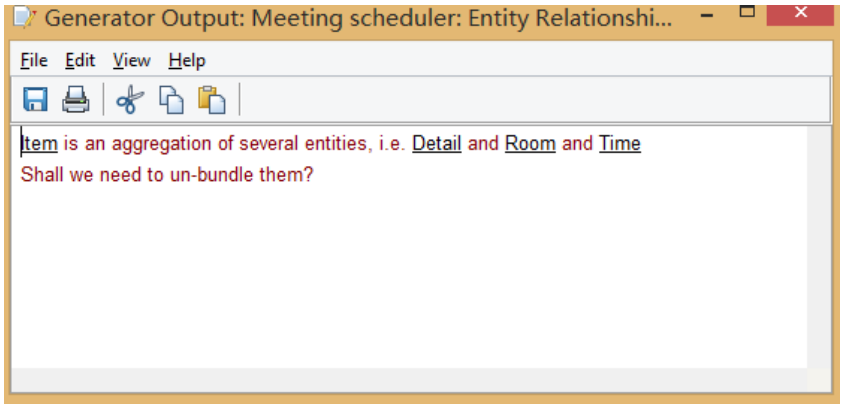

Figure 34. The result of aggregation redundancy detection

Some types of incompleteness are directly related to the properties of relationships. Since some relationship cannot be determined at the start of a project, a redundant property can be added to the relationships which may be not necessary in the model. Also we can use the Symbol Editor here to add a condition with the is*Redundant* property, and highlight the relationship which modelers are suspicious of the redundancy. The highlighted relationship is presented in a red dashed outline, as shown in Figure 35. The report is given in Figure 36.

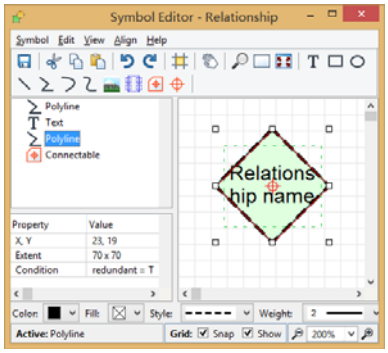

Figure 35. Symbol Editor on the Relationship

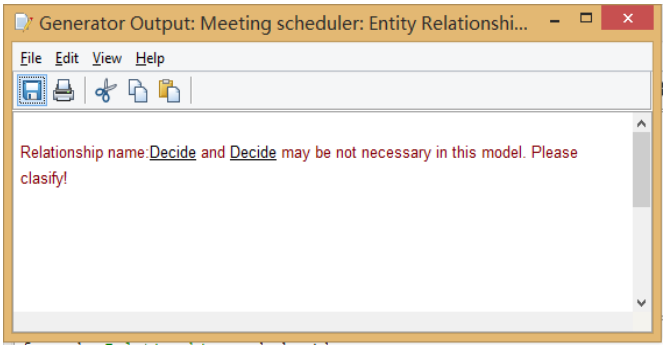

Figure 36. Redundant relationship detection

At last, MetaEdit+ also provides string and number commands functions to count how many entities are related to one relationship, which means that is possible to know whether a relationship is binary or ternary. Since all the relationships in this example is binary, the generation process is leaved out here.

6.2. **Incompleteness related with the properties of an entity**

As discussed in the previous section, the derivation business rules have an impact on the specification of entities and their attributes, such as the weakness of an entity, optional of an entity, multi-valued attributes and number of identifier of an entity. In this section, we demonstrate how such confusion can be detected and reported.

Since the weakness ER diagram has been extended with a Boolean property *isWeak*, the modelers can choose the weakness property when creating an entity in the modeling process, and create a highlight effect (e.g. add a red dashed outline on the weak entity) by using the Symbol Editor, which is shown as Figure 37. This highlight method imitates the modeling process introduced in Section 5.1.2.

The code is given in Figure 38.

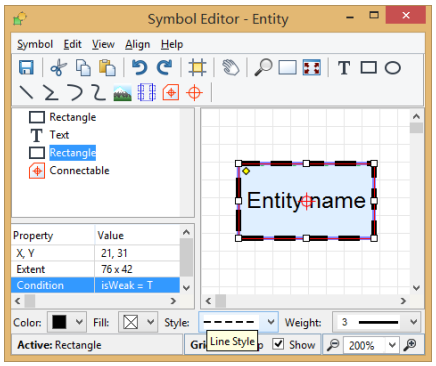

Figure 37. Symbol Editor on the Entity

```
Report '_detectWeakness'

foreach .Entity; orderby id
  {
    if :isWeak = 'T'
    then   id  ' is a weak entity.' newline

    do >In relationship.Relationship.Entity
     {
      'It depends on entity ' id newline
     }
     'Please check! '
    endif
  }
endreport
```

Figure 38. Code of weakness of entity detection

By this generator, each entity will be checked by this function, and all weak entities will be shown in a question format as shown in Figure 39.

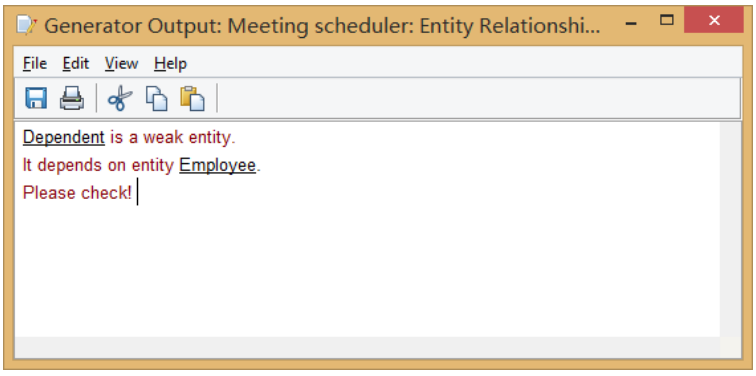

Figure 39. Result generated by *isWeak* function

In a similar way, the definition if an entity is optional or mandatory is also added to the Entity with the property *isOptional*. Also, by using the Symbol Editor, entities can be added with a circle mark when their *isOptional* value is true.

The result of optional entities detection is shown as Figure 40.

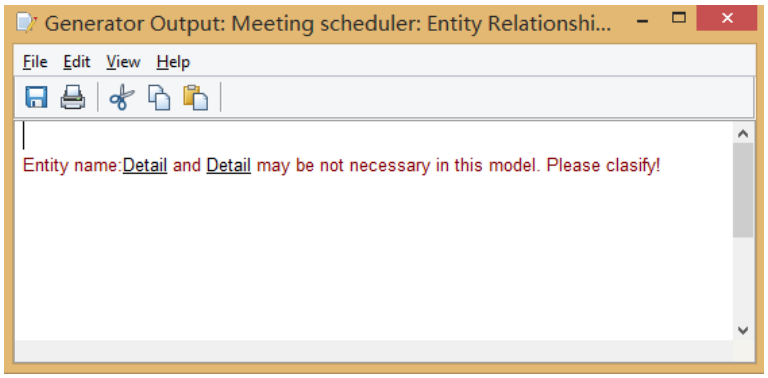

Figure 40. Result of optional entities check

These incomplete requirements about numbers of entities are similar with the optionality of the occurrence of an entity in a relationship. Therefore, I leave out the solution on this incompleteness problem.

6.3. **Incompleteness related to the property of an attribute**

As for the attribute, most of the information, such as the values and primary key, can be described in the property *Constraint*. The constraints can be NULL, NOT NULL, NOT NULL UNIQUE, NOT NULL PRIMARY KEY, as shown in Figure 41. NOT NULL indicates that the value of this attribute cannot be null. NOT NULL UNIQUE means there is only one value of this attribute which is opposite of a multi-valued attribute. The NOT NULL PRIMARY KEY options defines that one entity is identified by only this attribute. A blank content of this value means there are no constraints on this attribute which is also a signal that the constraints may be unclear at the start of modeling process.

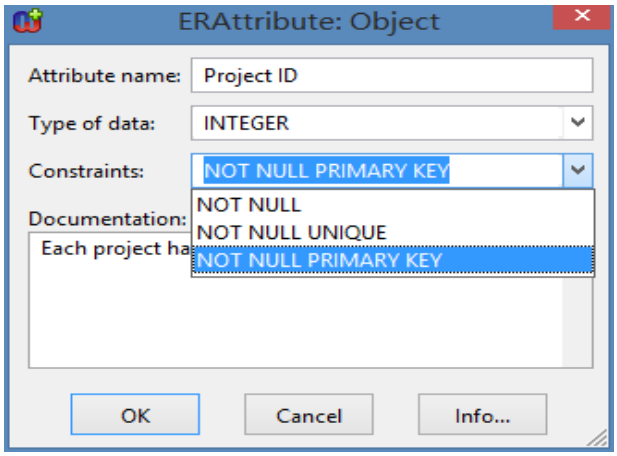

Figure 41. Constraints of an attribute

All the NOT NULL and NULL constraints will be treated as a feature of multi-valued attribute and marked with an dashed outline by using the Symbol Editor, which is shown as Figure 42. The generator will look through the model with necessary information and the code to detect each constraint is in Figure 43.

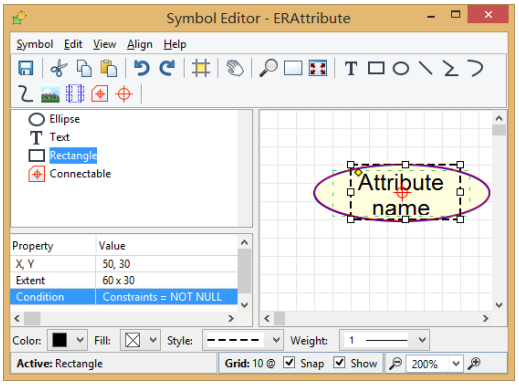

Figure 42. Symbol Editor with the Attribute

```
Report '_mulitvaluedAttribute'

foreach .Entity
{
    'Entity name: ' id
     newline
    do .ERAttribute
    {
        if :Constraints = ''
        then
        'Attribute name: ' id
        newline
        'Constraint type: NULL' newline
        'This attribute may be multivalued, is this necessary?'
        else ''
         endif

        if :Constraints = 'NOT NULL'
        then
        'Attribute name: ' id
        newline
        'Constraint type: ' :Constraints
        newline
        'This attribute may be multivalued, is this necessary?'
        else ''
         endif
        newline
    }
     newline
}
Endreport
```

Figure 43. Code of multi-valued attributes detection

The detecting result is shown as Figure 44.

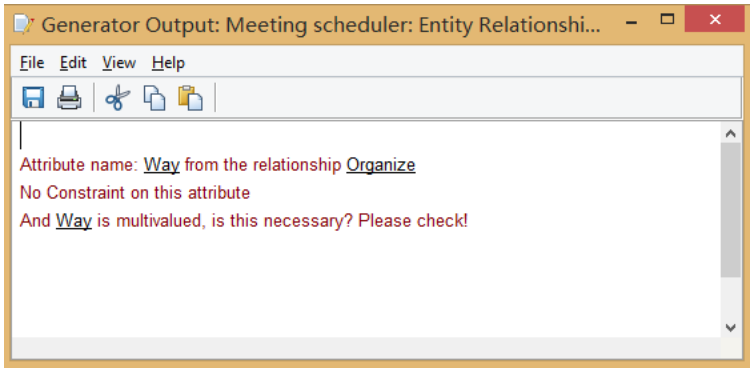

Figure 44. Result of the multivalued attribute generation

In a similar way, the attribute belonging problems can also be detect by the same code with a few changes. The result is shown as Figure 45.

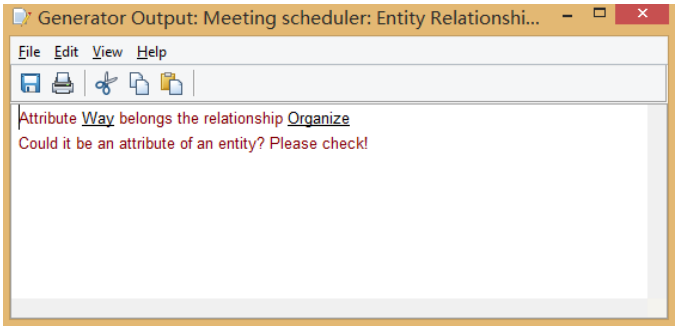

Figure 45. Result of attribute belonging detection

Besides, we add an underline mark by using Symbol Editor when the Constraint of an attribute is NOT NULL COMPOSITE PK, which shows the primary key attributes for all the entities, and we can add a underline on each of PK attribute by Symbol Editor. This constraint can also be used to detect the composite primary key of an entity and the result is shown as Figure 46.

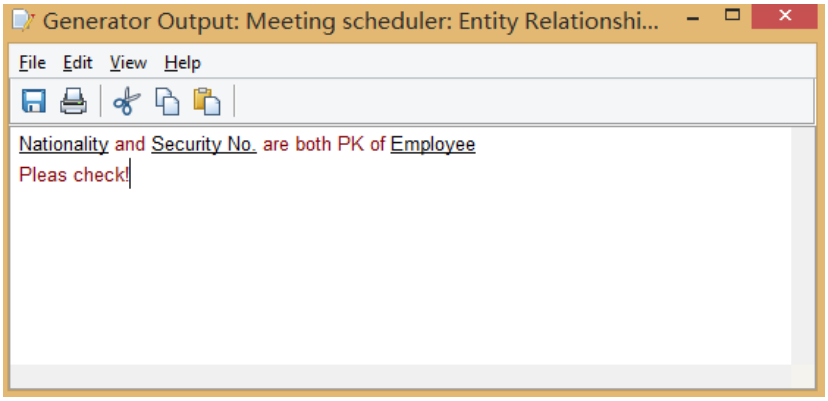

Figure 46. Result of the composite identifier in the model

7. Discussions

This section discusses the generated reports, and further classifies the reports into different groups to analyze the unknowns and to suggest the follow-up actions.

7.1. Classification of incompleteness

Since some missing information can be identified and represented during the modeling process, while some other unclear information like model redundancy is hard to detect on the basis of the metamodel specifications, I divided the automatically generated reports into three categories, i.e. detected missing information, suspicious issues, and unsolved (unshown) problems, as shown in Table 3.

Detected missing information	connective in a relationship
	relationship redundancy
	optional entity (number of entities)
Suspicious issues	overlapping or disjoint hierarchy
	aggregation hierarchy
	weak entity
	multi-valued attribute
	attribute belonging
	composite identifier
Unshown problem	relationship degree
Unsolved problem	exclusive or inclusive of relationships

Table 3. Classification of Incompleteness

As seen in the Table 3, we captured the missing information on optional (number of) entities, relationship cardinality and redundancy. The weakness entities, amount of attributes values, number of attributes identifiers, attribute belonging, aggregation hierarchy and supertype-subtype relationship (disjoint or overlapping) are marked as suspicious issues rather than incomplete problems, because we cannot determine whether the information is missing or not when we found it. Because of the limitation of the example, the incompleteness in relationship degree has not been shown. At last,

the exclusive or inclusive of relationships cannot be detected and solved by MetaEdit+.

7.2. **Solutions**

By including the unknown issues into the graphical symbol definition in the Symbol Editor in the metamodeling process, all the missing information can be highlighted in the model as shown in Figure 47 with red color. For example, the ambiguous relationship connectivity is focused by the rectangle outline of the question mark. The optional entity *Date* has a circle on its left as a mark of the unnecessary entity, and the redundant relationship *Propose* is shown with a dashed outline in the model. As for model verification performance [Carson, 2002], modelers can locate the incompleteness quickly in the modeling process with these highlighted symbols.

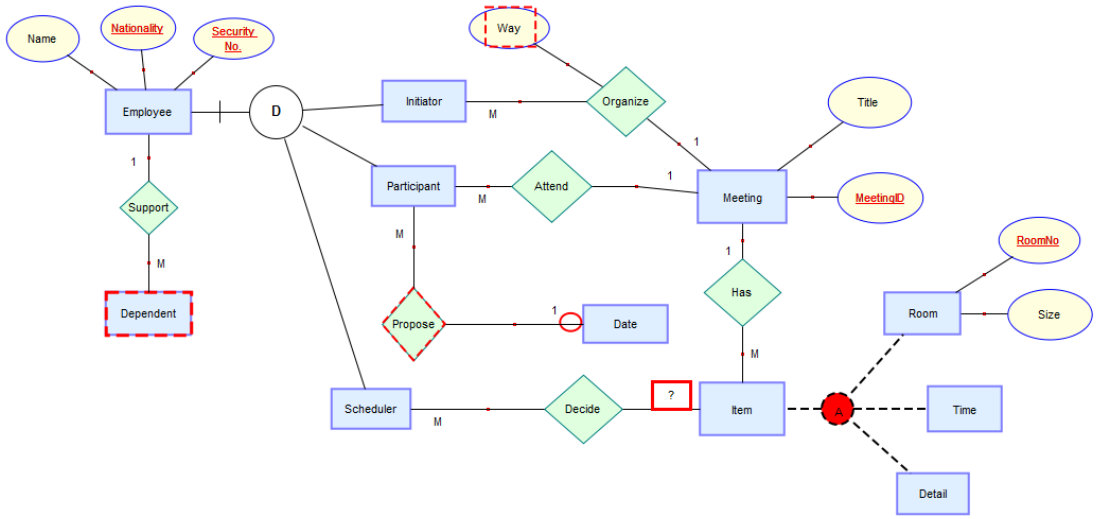

Figure 47. Meeting Scheduling ER model with Incompleteness highlight

Meanwhile, most of the suspicious issues are highlighted on the ER model, such as the dashed outline of the weak entity *Dependent* and multi-valued attribute *Way*, yellow background relationship Aggregation with dashed line, O/D on the supertype-subtype and underline on the PK attribute of several entities. However, not all the suspicious issues discussed in literature can be shown directly in the ER model. For example, the attribute belonging problem is not represented directly in the model. Therefore, we need to generate a report to show this suspicious incompleteness in an easy-understand way after the modeling process.

Generally, all the unclear or unsure information is caused by the lack reality perceived from requirements engineers in the problem domain. When perceived information is incomplete, the modeler may construct an improper relationship between objects or assign an improper attribute to an object. With the metamodeling specification, incompleteness detection and report by MetaEdit+, the missing

information and suspicious issues will be generated in natural language and sent back to the requirements analysts. Based on the results, requirements analysts will produce a check on the related business rules which is provided by the ER elements link model. Some of the missing information and suspicious issues may be clarified by checking the business rules, while others can capture the incompleteness in the requirements and a discussion among the stakeholders will be hold to perfect the business rules. For example, after we detect that *Dependent* is a weak entity by the metamodel specification process, the result will be sent to the requirements analysts for a check on the Derivation type of business rules. The key words are *Employee*, *Dependent* and related information about leaving or accident about one *Employee.* If we find some business rules about an *Employee* leaving the company and his/her dependent information will be removed, the *Dependent* is weak, and otherwise it will not be a weak entity. If we cannot find such information, there could be some incompleteness in the business rules and a further discussion is needed to modify both the requirements and the model later.

Meanwhile, with the string and number commands counting functions, degree of a relationship can be detected on the basis of the metamodel specified in this thesis and defined as the unshown problem because of the limitation of the example. However, the exclusive or inclusive relationships cannot be detected neither because it is too complicated to add the *isExclusive* or *isInclusive* property to the ER metamodel nor models. It is not the disability of MetaEdit+ but the limitation of ER metamodel in the definition and representation aspect.

Solutions on the unsolved problem are threefold. Firstly, improving the quality of requirements at the elicitation stage by writing excellent requirements. Normalizing the requirements writing style, documenting the appropriate details and avoiding ambiguity can improve the requirements and decrease the incompleteness [Wiegers and Beatty, 2013]. Secondly, increasing the communications with the stakeholders. One of the most important part in software development cycle is to focus on the feedback from the modelers and increase the discussion among the stakeholders. Communication is not simply a matter of putting requirements on paper and tossing them over a wall. It involves ongoing collaboration with the team to ensure that they understand the information you are communicating [Wiegers and Beatty, 2013]. Thirdly, a software development model with communication at all levels of system hierarchy is appropriate way to minimize the risk of all possible problems.

Traditional software development models, such as waterfall development model, suggests a systematic, sequential approach to software development that begins with customer specification of requirements, and requires modelers to plan and schedule all

of the process activities before starting work on them [Pressman, 2007]. However, it is often difficult for the customer to state all requirements explicitly before a project starts. Therefore, waterfall developments model has difficulty accommodating the natural uncertainty that exists at the beginning of many projects [Pressman, 2007].

Nowadays, software work is fast paced and subjected to a never-ending stream of changes. The traditional development models are often inappropriate for such work. Therefore, models which can start with unclear requirements and focus on the communications in the development cycle are seen as the solutions, e.g. agile development model.

Agile development model encourages rapid and flexible response to change by short timeboxed iterations with adaptive and evolutionary refinement of plans and goals [Larman, 2004], and one of the key issues which is stressed by the agile philosophy is communication and collaboration between team members and practitioners. As one of the most widely used agile process, Extreme Programming (XP) sets the communication and feedback as one of its core value, and the best way of communication is face-to-face and avoid formal specification. Another agile development model is Scrum model, which emphasizes the use of a set of software process patterns that have process patterns that have proven effective for projects with tight timelines, changing requirements, and business criticality. Each process pattern defines a set of development tasks and allows the Scrum team to construct a process that is adapted to the needs of the project. In practice, Scrum development model advocates the project team to working in a common project room and self-directed teams [Larman, 2004]. Continuous discussion and communication on the software development is kept through a daily stand-up meeting [Larman, 2004].

Therefore, agile development model is suitable for this incompleteness detection and results generation methodology.

8. Conclusion

Inspired by the MBRM framework, we present an approach to capturing and reporting requirements incompleteness through a metamodeling-based detection process.

By using the metamodel specification, we can build generator in MetaEdit+ to detect and report the incompleteness in an ER model. The generated report can be divided into two groups, i.e. missing information and suspicious issues. All the results will be sent to the requirements analysts for a further check on business rules. Some of the unclear information can be clarified by checking, while others can show up the incompleteness in business rules. Discussions and communications will be raised among stakeholders to improve the related requirements.

Because of the limitation of the example and metamodel representation, some of the incompleteness cannot be shown or detected by the MetaEdit+. There are three ways to decrease the possibility of the undetected incompleteness, i.e. improve the quality of requirements specification, increase the discussion among project members, and apply appropriate development models with more focus on communications.

In summary, the contribution of my thesis is twofold. Firstly, with the help of Symbol Editor, modelers can capture and locate the missing information and suspicious issues in the ER model, and by using the automated detection and report function, all the incompleteness problems can be sent to the requirements analysts in the report format as feedbacks. Secondly, in my thesis, the metamodeling detecting process is applied to the ER model, and the basic theory based on the MBRM framework can be also used on other modeling method, such as Class Diagram, State Transform Diagram. Moreover, the limitation of metamodeling representation on polishing of the MetaEdit+ generators form the new direction of future work of my thesis.

References

[Ebert and De Man, 2005] Christof Ebert and Jozef De Man: Requirements uncertainty: Influencing factors and concrete improvements. *ICSE'05*, May 15-21, 2005

[Parnas, 1979] Parnas, D.L.: Designing Software for Ease of Extension and Contraction. IEEE Trans. On *Software Engineering*. Vol.5, No.2, 1979

[Zhang et al., 2014] Zheying Zhang, Peter Thanisch, Jyrki Nummenmaa, Jing Ma: Using a metamodel to detect missing requirements, *20th International Conference, ICIST 2014*: 248-259

[Wiegers, 1999] Karl E. Wiegers, Writing quality requirements. *Software Development*, May 1999.

[Carson, 1998] Ronald S. Carson, Requirements Completeness: A Deterministic Approach, *Incose International Symposium, 1998*, 8(1): 738-746

[Zowghi & Gervasi, 2006] The Three Cs of Requirements: Consistency, Completeness, and Correctness, *8th International Workshop on Requirements Engineering*: Foundation for Software Quality, 2006

[Wan-Kadir and Loucopoulos, 2003] W.M.N. Wan-Kadir, Pericles Loucopoulos, Relating evolving business rules to software design. *Department of Computation, University of Manchester Institute of Science and Technology (UMIST),P.O. Box 88, Manchester M60 1QD, UK,* Available online 27 November 2003

[Thanish et al., 2013] Peter Thanisch, Tapio Niemi, Jyrki Nummenmaa, Zheying Zhang, Marko Niinimäki, Pertti Saariluoma, Incompleteness in conceptual data modelling. *Communications in Computer and Information Science* Volume 403, 2013, pp 159-172

[Kelly and Lyytine, 2005] Steven Kelly and Kalle Lyytinen, MetaEdit+: A fully configurable Multi-User and Multi-Tool CASE and CAME environment, *Department of Computer Science and Information Systems, University of Jyv~iskyla*, 2005

[Kotonya and Sommerville, 1998] Gerald Kontonya and Ian Sommerville, *Requirements Engineering: Process and Techniques*. Wiley, 1998.

[Wiegers and Beatty, 2013] Karl Wiegers and Joy Beatty, *Software Requirements*. 3rd Edition, Best practices by Microsoft, 2013.

[IEEE std. 610.12, 1990] Institute of Electrical and Electronics Engineers: IEEE Standard Glossary of Software Engineering Terminology (*IEEE std. 610.12-1990*). IEEE Computer Society, New York, 1990.

[Pohl and Rupp, 2011] Klaus Pohl and Chris Rupp, Requirements Engineering Fundamentals: A Study Guide for the Certified Professional for Requirements Engineering Exam, 1st Edition, 2011.

[Cohn, 2010] Mike Cohn, Succeeding with Agile: Software development using Scrum. *Upper Saddle River, NJ: Addison-Wesley*, 2010.

[Gorton & Ernst, 2014] Neil A. Ernst, Ian Gorton, Using AI to model quality attribute tradeoffs. *AIRE 2014*: 51-52

[Rosca et al., 1999] Rosca, Greenspan, Feblowitz, and Wild, A decision support methodology in support of the business rules lifecycle. Paperpresented at the International Symposium on Requirements Engineering (*ISRE'97*), Annapolis, MD.

[Herbst, 1996] Herbst, Business rules in system analysis: A meta-model and repository system. *Information Systems*, 21(2), 147-166.

[Halle, 1994] Halle, Back to business rule basics. *Database Programming and Design* (October 1994), 15-18.

[Sommerville, 2011] Ian Sommervile, *Software Engineering*. 9th Edition, Addison-Wesley, 2013.

[McManus & Hastings, 2005] McManus, H. L. and Hastings, D. E., "A Framework for Understanding Uncertainty and its Mitigation and Exploitation in Complex Systems," *IEEE Engineering Management Review*, Vol. 34, No. 3, Third Quarter 2006, pp. 81-94. (Originally published in the proceedings of the Fifteenth Annual International Symposium of the INCOSE, Rochester, NY, and July, 2005.

[Chalupnik et al., 2009] Marek J. Chalupnik, David C. Wynn and P. John Clarkso, Approaches to mitigate the impact of uncertainty in development processes, International conference on Engineering Design, *ICED'09*, 24-27, August, 2009.

[Bstieler, 2005] Bstieler, L. The moderating effect of environmental uncertainty on new Product Development and time efficiency. *Journal of Product Innovation Management*, 22(3), 267-284, 2005.

[Salay et al., 2012] Rick Salay, Marsha Chechik, and Jennifer Horkoff, Managing requirements uncertainty with Partial Models. University of Toronto, Dept. of Computer Science, Report 2012.

[Bonarini, 2010] Andrea Bonarini, Uncertainty management – Knowledge engineering course. Politecnico Di Milano, Dept. of Electronics, 2010-2011.

[Shiv and Doraiswamy, 2012] Premi Shiv, Premanand Doraiswamy, 50 Top IT Project Management Challenges, IT Governance Ltd, 2012

[Zhang et al., 2013] He Zhang, Juan Li, Liming Zhu, Ross Jeeryb, Yan Liud, Qing Wangc, Mingshu Lic, Investigating Dependencies in Software Requirements for Change Propagation Analysis, Preprint submitted to *Information and Software Technology,* June 17, 2013.

[Firesmith, 2005] Are Your Requirements Complete? *Journal of Object Technology* 4(1): 27-44 (2005)

[Khan et al. 2008] S. S. Khan, P. Greenwood, A. Garcia, and A. Rashid, On the impact of evolving requirements-architecture dependencies: An exploratory study. In Proceedding of *20th International Conference on Advanced Information Systems Engineering (CAiSE'08)*, volume LNCS 5074, pages 243-257, Montpellier,France, Jun. 2008.

[Carlshamre and Regnell, 2000] P. Carlshamre and B. Regnell. Requirements lifecycle management and release planning in market-driven requirements engineering processes. In Proceedings of *11th International Workshop on Database and Expert Systems Applications (DEXA'00), IEEE*, pages 961-965, London, UK, Sept. 2000.

[Yu, 1997] Eric S. K. Yu, Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering, Faculty of Information Studies, University of Toronto Toronto, Ontario, Canada M5S 3G6, 1997

[Bézivin, 2005] Jean Bézivin, On the unification power of models. *Software and Systems Modeling* 4(2): 171–188, May 2005.

[Rothenberg, 1989] Jeff Rothenberg, The nature of modeling. Prepared for: *Defense Advanced Research Projects Agency*, November 1989.

[Génova, 2009] Gonzalo Génova, Modeling and metamodeling in Model Driven Development - What is a metamodel: the OMG's metamodeling infrastructure, *Knowledge Reuse Group, Universidad Carlos III de Madrid*, Warsaw, May 14-15th, 2009.

[Brinkkemper, 1990] Formalisation of information systems modeling. *Brinkkemper, JN*., 1990

[metamodel.net, 2014] metamodel.net, What is metamodel. Available at: http://metamodel.net/ (Accessed: 2014).

[Tolvanen et al., 2007] Juha-Pekka Tolvanen, Risto Pohjonen, Steven Kelly, Advanced Tooling for Domain-Specific Modeling: MetaEdit+, 2008.

[Fidalgo et al., 2012] Robson Do Nascimento Fidalgo, Elvis Maranhão De Souza, Sergio España, Jaelson Brelaz De Castro1and Oscar Pastor, EERMM: a metamodel for the Enhanced Entity-Relationship Model. In: *Proc. of 31st International Conference*, *Lecture Notes in Conceptual Modeling* (2012), Springer, p515.

[Carson, 2002] John S. Carson: Model Verification and Validation, Proceeding of the *2002 Winter Simulation Conference*, 2002

[Pressman, 2007] Software Engineering: A Practitioner's Approach, International version, 2007

[Larman, 2004] Craig Larman: Agile and Iterative Development: A Manager's Guide, Addison-Wesley Professional, 2004