RGML: A Specification Language that Supports the Characterization of Requirements Generation Processes

by

Ahmed Samy Sidky

Thesis submitted to the Faculty of
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
Computer Science and Applications

James D. Arthur, Chair
Osman Balci
Scott McCrickard

July, 2003
Blacksburg, Virginia

Keywords:
Requirements Generation Process, Requirements Specification Language,
Specification Language, Process Description Language, Markup Language

RGML: A Specification Language that Supports the Characterization of Requirements Generation Processes

Ahmed Samy Sidky

ABSTRACT

Despite advancements in requirements generation models, methods and tools, low quality requirements are still being produced. One potential avenue for addressing this problem is to provide the requirements engineer with an interactive environment that leads (or guides) him/her through a structured set of integrated activities that foster "good" quality requirements. While that is our ultimate goal, a necessary first step in developing such an environment is to create a formal specification mechanism for characterizing the structure, process flow and activities inherent to the requirements generation process. In turn, such specifications can serve as a basis for developing an interactive environment supporting requirements engineering.

Reflecting the above need, we have developed a markup language, the Requirements Generation Markup Language (RGML), which can be used to characterize a requirements generation process. The RGML can describe process structure, flow of control, and individual activities. Within activities, the RGML supports the characterization of application instantiation, the use of templates and the production of artifacts. The RGML can also describe temporal control within a process as well as conditional expressions that control if and when various activity scenarios will be executed. The language is expressively powerful, yet flexible in its characterization capabilities, and thereby, provides the capability to describe a wide spectrum of different requirements generation processes.

ACKNOWLEDGMENTS

First and foremost I am grateful and thankful to Allah (God) who blessed me with guidance, good health, and good friends that supported me all through my life. I thank Him for giving me patience and helping me through all the tough times and especially, during the writing of this thesis. I am also forever grateful to my beloved parents and sisters, who helped me through every step of my life till I have reached where I am today.

Additionally, I am grateful to my professor Dr. James Arthur. His direction, motivation and inspiration lead to the accomplishment of my thesis. Whenever I needed Dr. Arthur, he was *always* there for me. Having his door always open for me, is something I will always cherish. Also I am grateful to Dr. Sara Thorne-Thomsen for help and advice throughout the writing of this thesis.

Furthermore I am grateful to my committee members Dr. Osman Balci and Dr. Scott McCrickard for their valuable help and contribution in this research.

Moreover, I am grateful to the wonderful Muslim community in Blacksburg that had a significant impact on my life. The Muslim community especially MSA, VTMUGS and MBK provided me with moral, spiritual and emotional advice and support and were there for me whenever I needed them.

Last but not least I am grateful to the computer science department, faculty and staff and fellow graduate students who were an essential part of my overall learning experience.

Yet again, all thanks and gratitude goes to Allah only.

(El Hamd el Allah)

TABLE OF CONTENTS

LIST OF EXAMPLES

LIST OF FIGURES

1. Introduction

A requirements generation processes encompasses all the activities that lead to the synthesis of a set of requirements for a particular system. These activities include problem analysis, needs generation, requirements elicitation, requirements analysis, requirements verification and validation, and requirements specification. This thesis presents the Requirements Generation Markup Language (RGML), a simple and easily understood mechanism for specifying and describing the requirements generation process. It also provides a discussion of existing requirements generation processes to highlight the advantages of RGML, as well as a description of RGML's objectives, syntax, and semantics. The material presented illustrates how RGML inherits salient characteristics of existing specification and process description languages and utilizes them to give the requirements engineer simple, but complete guidelines for the requirements generation process. RGML is the first of four components needed to build an interactive tool for guiding the requirements engineer through the requirements generation process.

1.1 Developments in the field of Requirements Engineering

Since the current state of requirements engineering is the main motivation behind the development of RGML, it is important to review the developments in this field to understand RGML's place within it. Requirements engineering is a field within the larger field of software engineering. The current state software engineering, and especially requirements engineering, has reached is a prime motivator for the development of RGML. Good software engineering practices are necessary for the development of any medium to large scale software system. Royce's [Royce 1987] introduction of the waterfall model in 1970 brought out the importance of software engineering as a discipline to structure the activities performed previously in an ad-hoc manner. The Waterfall Model is the first attempt to organize the process of software development. Royce divides the software development lifecycle into five main activities: Requirements Analysis, Design, Implementation, Integration and Testing, and Maintenance. Following the Waterfall Model, several other models evolved; these include the spiral, prototype, incremental and evolutionary models. Each of these models approaches the software

development cycle from different perspectives. In each of these models, requirements analysis plays an important role as an initial development phase. As envisioned during its inception, requirements analysis was composed of a series of activities that relate to the gathering and analysis of requirements from the customer. Although "first" in the sequence of phases, Requirements Analysis has been last in line for re-examination and refinement by the software engineering community [Sidky 2002]. Only recently are researchers realizing that the Requirements Analysis phase is the most important phase in the software development lifecycle. Consequently, this phase is receiving more attention as researchers are looking for new and innovative ways to improve the activities within this phase [Davis 1993].

1.1.1 Realizing the importance of requirements

Since a system is only as good as the requirements from which it is developed, it is crucial for the requirements engineer to gain a firm understanding of the customer's stated, as well as implied, requirements. Recognition of the relationships between the quality of a product and the quality of the requirements used to develop it constantly reinforces the importance of and need for a well-defined, effective set of requirements engineering activities. So far, however, the success rates for development efforts are abysmal. According to the 1995 assessment of the Standish Group five, of the top eight reasons why projects fail are related to the requirement generation process – incomplete and/or erroneous requirements, lack of user involvement, unrealistic customer expectations, and requirements volatility [Standish 1995]. Brooks [1987] also states that:

> *The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.*

With the realization of the importance of requirements engineering, the originally coined "requirements analysis" phase has evolved into "requirements engineering." The Software Engineering community has refined requirements engineering and divided it into five distinct activities. Reflecting a more current view of requirements engineering, Pressman [2001] describes those activities:

- **Requirements Elicitation**: In this activity, requirements engineers elicit software requirements from the customer or derive them from system requirements.
- **Requirements Analysis**: The requirements engineer usually performs this activity before the customer commits to the actual development process. The customer assesses the acceptable level of risk regarding the completeness, correctness, and technical and cost feasibility.
- **Requirements Specification**: In this activity, the requirements engineer documents and expresses the requirements elicited and analyzed from the preceding activities in the form of a formal document, often referred to as the Software Requirements Specification (SRS).
- **Requirements Verification and Validation**: This activity ensures that the requirements elicited and specified in the SRS adhere to the customer's needs or the high level requirements. The requirements engineer tests the specified requirements for adherence to pre-defined quality attributes.
- **Requirements Management**: This activity pervades the entire requirements engineering process and facilitates communication effectively, in case any changes are made. Requirements Management is "a process that establishes and maintains agreement between the customer and the project team on the changing requirements of the system" [Leffingwell 2000].

Over the past few years, researchers have focused on each of these five activities with the intention of developing tools and methodologies for each. Because the integration of these activities has not been an initial priority, each activity became an area of research of its own. Today, however, along with those independent research activities,

complementary research on integrating those activities into a *comprehensive* requirements generation process or model has begun.

1.1.2 The current situation of requirements engineering.

Today, requirements generation models and frameworks, which provide a 'skeleton' – an overarching structure that guides the process progression, are popular. Requirements generation models take the five main activities defined earlier, organize them, and put them into phases. Several of these models have been developed, and each is based on a particular approach and designed with a specific set of methods in mind. Among those requirements generation models are the Requirements Generation Model (RGM) [Arthur 1999], the Requirements Triage [Davis 1999], the Win-Win Spiral Model [Boehm 1988], and the Knowledge-Level Process Model [Herlea 1999].

In spite of all the advancements in requirements engineering, however, there are still major problems facing the requirements and software engineering communities. Observations from practitioners and researchers continually indicate "poor" quality requirements, frequently changing requirements, and elicited requirement sets that are often incomplete. Moreover, observations related to requirements generation processes indicate that the quality of the requirements generated from the process are unpredictable; the same requirements engineer can use a requirements generation process to produce a good set of requirements and then produce a "poor" set of requirements with the same process.

While these observations are not themselves the problems, they certainly indicate the existence of problems in the field of requirements engineering. They also underscore the need to identify with some accuracy the root cause(s) underlying the above mentioned observations. Section 1.2 provides a list of these causes and an explanation of the issues behind each of them.

1.2 Problem Statement

This section provides a reflection on the above observations and identifies several issues related to those observations. Those issues are then reformulated to provide a Problem Statement that is addressed by the research described in this thesis.

The observations in Section 1.1.2 of this Chapter indicate the existence of problems and problem areas within the field of requirements engineering. Pinpointing the actual cause(s) of such problems is not always easy. However, some of the more identifiable ones include:

- **Understandability of the requirements engineering processes**: A large number of those who are responsible for the requirements engineering process in projects have little or no understanding of the science of requirements engineering.
- **Lack of process structure enforcement**: Even if the requirements engineer understands the requirements engineering process, he or she can optionally choose which pieces to employ or ignore. This situation creates a problem, because a good set of requirements depends on the skills of the requirements engineer and adherence to the specified process.
- **Process mismanagement**: There are cases where the requirements engineer is competent and skillful, but where management oversight leaves much to be desired. In some instances management simply ignores what the requirements engineer is doing; in others, the managing supervisor places unnecessary constraints on the requirements engineer. Poor management is a major reason for the failure of requirements engineering efforts, especially if the management itself is not familiar with the process.
- **Ad-hoc composition of sub-processes**: In many cases, the requirements engineer does not follow a requirements generation process and develops his or her own process composed of an ad-hoc collection of sub-processes, activities, and tools which are often incompatible with each other.

Reflecting on the four problems noted above, two issues emerge which, if adequately addressed, can mitigate the detrimental impact of these problems. These two issues are **lack of understanding** by the requirements engineer of the requirements generation process, and the **lack of guidance** given to the requirements engineer while executing the process. One approach to addressing these two issues is to formally describe the requirements generation process and use that description in a way that guides and assists the requirements engineer through the execution of the process and to increase his/her potential of understanding the process.

Developing such a mechanism to formally describe the requirements generation process introduces a number of challenges, some of these are:

- How to describe the structure of the process
- How to express the steps involved in an activity
- How to express the flow of execution of the process
- How to describe when and where to instantiate tools and applications in the process
- How to express the conditions within the process
- How to describe the wide variety of requirements generation processes using one mechanism.

These challenges must be considered in the creation of any language that supports the specification and description of a requirements generation process. The next section presents an approach to defining the solution and places it in the context of an interactive user environment that supports the requirements engineer in the requirements generation process.

1.3 The Solution Approach

The solution approach described in this section incorporates both long-term and short-term objectives. The long-term objectives enable us to set the stage for our goal, i.e, providing a mechanism for *describing* the requirements generation process.

1.3.1 The Overall Solution Approach

This research envisions an interactive requirements generation environment as the final solution to the overall "lack of guidance" problem. This environment is designed to guide the requirements engineer through an integrated set of structured activities, which reflect a unified, well-defined requirements generation process. The environment is envisioned having three main components:

- **A Specification Language** that can formally describe the characteristics of any requirements generation process.
- **An Editor** that is divided into two parts: a low-level editor the designer uses to capture the process, and a high-level editor the requirements engineer uses to tailor the process.
- **An Interactive Application** that interprets the formally specified requirements generation process and visually guides the requirements engineer through the execution of the process.

The interactive environment addresses the previously identified problems inherent to requirements engineering. The environment helps requirements engineers understand the process, because the process is presented to them visually in a manner that they can follow. The environment also assists the less experienced engineers by guiding them step by step through what they should do and by providing follow-up to the execution of the process. This also has the benefit of reducing the effect of poor management on the execution of the requirements engineering processes. Since the environment provides the requirements engineer with a pre-defined requirements generation process to follow, he

or she is not likely to compose ad-hoc sub processes or ignore important components within the process. Chapter 6 provides more details about the interactive environment.

The challenge of developing the first component of this environment, the specification language, can be viewed as a problem of **how to specify and describe the requirements generation process in a way that guides and assists the requirements engineer through the process.** Developing that first component is the focus of our research, and subject of this thesis.

1.3.2 The First Component – The Specification Language

The first component that must be developed before any other component in the environment can function is the specification language. The specification language is to support the formal specification and characterization of the requirements generation processes to be used within the environment. This research, therefore, addresses and resolves the problems and challenges associated with devising this formal language.

The ultimate goal of this language, besides being the blueprint for the interactive environment, is to provide guidance to the requirements engineer through direct interpretation of the formal specifications of the process in the language. The specification mechanism must be flexible enough to capture the variations among requirements generation models. These include variations in process structures, artifacts (produced and consumed), and actions. It must also be simple and easy to understand, yet formal enough to admit to programmatic interpretation. The ***Requirements Generation Markup Language (RGML)***, described in this thesis, is just such a language. *RGML* is flexible and expressively powerful and supports the detailed characterization of those elements necessary to, and found within, requirements generation processes.

RGML is a powerful language that permits the specification of:

- protocols and guidelines that lead the requirements engineer through the requirements generation process,

- production and consumption of artifacts throughout the generation process and the association of templates with artifacts,
- automatic instantiation of software applications for the production of artifacts,
- selective control flows and activities based on temporal and conditional specifications, and
- multiple execution scenarios for a given activity.

Although its primary function is to support the formal description of a requirements generation process, RGML has additional benefits. It pulls together under one umbrella the major capabilities and components needed when describing a requirements generation process, thereby increasing the level of awareness of what is necessary to develop a new requirements generation process. RGML provides a standard basis from which all requirements generation processes can be expressed, thereby facilitating systematic comparisons between different processes. In addition, as stated earlier, RGML is the first step towards building an interactive environment that would guide requirements engineers through requirements generation processes.

In summary, RGML is the first step towards solving one of the major problems facing requirements engineering. That is, RGML directly addresses the problem of **specifying and describing the requirements generation process in a way that guides and assists the requirements engineer through the process.** As the discussions in the following chapters demonstrate, RGML addresses the issues of structure, steps, flow, tools, conditions, and other aspects of requirements generation processes through its rich set of constructs and its unique design.

Chapter 2 provides a discussion of work done in other fields, including specification languages, process description languages, and markup languages and their relationship to RGML. Chapter 3 provides an overview of the structure of RGML, while Chapters 4 and 5 concentrate on describing the details of the language syntax. Chapter 6 provides a discussion of the relationships of RGML to the interactive requirements environment and a conclusion to this thesis.

2. Background

Chapter 1 discusses the importance of the contribution of the newly developed RGML to solving a number of problems in the field of requirements engineering. This chapter provides background information about the different developments in the areas that have had a major influence on the design and the objectives of RGML. These areas include specification languages, process description languages, and markup languages. This chapter also identifies the characteristics RGML inherits from each of these areas.

2.1 Specification Languages

This section of Chapter 2 concentrates on defining specification languages while mentioning various specification languages within the sub-fields of Computer Science, because RGML inherits certain characteristics of these languages. In addition, this section also provides a discussion of the differences between RGML and specification languages within the field of requirements engineering in order to highlight the contribution RGML is making to the field.

2.1.1 Definition of a Specification Language

Since RGML inherits some characteristics of specification languages, the definition of a specification language is important to the understanding of the development of this new requirements process markup language. A specification language is a language that can formally capture the salient characteristics of a specific domain. However, this definition is somewhat generic and vague, because an accurate definition of a specification language depends on the domain the language intends to capture. In the field of computer science, there are a large number of specification languages which address the needs of a number of different fields and, therefore, have different definitions. Some specification languages, such as Albert II [Du Bois 1995], the Problem Statement Language (PSL) [Teichroew 1982], the Requirements Specification Language (RSL) [Frincke 1992], and the Two-Level Grammar (TLG) [Bryant 1991],

have an objective of formally capturing requirements. Other specification languages, such as the Unified Modeling Language (UML) [OMG 1999] and Specification and Description Language (SDL) [Saracco 1989], have an objective of formally capturing system design. Another group of specification languages, the so-called Hardware Description Languages, including Verilog [Thomas 1991] and the Very High Speed Integrated Circuit Hardware Description Language (VHDL) [IEEE 1993], are designed to formally capture the specification and implementation of hardware. The list of specification languages continues to grow, because in any field of computing a specification language can be developed to formalize certain aspects within that field. Because of the different elements and characteristics of specification languages and their importance to the development of RGML, the next section concentrates on explaining in depth two of the specification languages designed to capture requirements.

2.1.2 Requirements Specification languages

This section provides descriptions of two specification languages, which have influenced the development of RGML. The descriptions are intended to convey the nature and characteristics of specification languages designed to capture the specification of requirements. A clear understanding of these requirements specification languages is necessary before proceeding to a discussion of the relationship between them and RGML in Section 2.1.3 of this chapter.

The first language, RSL, which shares several characteristics with RGML, has the objective of precisely describing the external structure of the objects and operations in a software system [Frincke 1992]. The operations and objects of a system originate from the set of requirements the requirements engineer gathers about the software system, but RSL does not describe a system by formalizing these requirements. Instead, the requirements engineer analyzes the set of gathered requirements, and then he or she uses RSL to capture the specification of the system by utilizing two models: the *object-oriented model* and the *operation-oriented model*. The *object-oriented model* views the system from the perspective of objects (i.e., data) and the *operation-oriented model* views the system from the perspective of operation (i.e., functions). Since these two models

provide differing views of the system, RSL uses them as complements to each other. Example 1 shows how RSL defines an object. Example 2 shows how RSL defines an operation.

```
object name [is]
      components: composition expression defining subobjects;
      [operations: list of applicable operations;]
      [equations: formal equational specification;]
      [description: comment or entity name;]
      [user-defined attributes: comment or entity name;]
end [name];
```

Example 1. How RSL defines an object

```
operation name [is]
      [components: composition expression defining suboperations ;]
      inputs: list of objects;
      outputs: list of objects;
      [preconditions: formal predicate on inputs;]
      [postconditions: formal predicate on inputs and outputs;]
      [description: comment or entity name;]
      [user-defined attributes: comment or entity name;]
end [name];
```

Example 2. How RSL defines an operation

The two proceeding examples illustrate how the requirements engineer uses a specific set of components and attributes with a fixed format, in RSL, to describe any *object* or *operation* within the system. This degree of standardized formal specification eliminates any ambiguity that can arise while reading the system's objects and operations. In addition, this formal specification creates a mechanism which the requirements engineer can use to formally analyze the operations and objects of the system.

The second language, which requirements engineers can use to formally specify requirements, and which RGML also shares some common characteristics with, is TLG. TLG (also called W-grammar) was originally developed as a specification language for programming language syntax and semantics. Bryant's approach [Bryant 1991] uses a Two-Level grammar to apply techniques from formal methods to the development of a specification methodology based on natural language. Devising a technique that specifies

requirements in a natural language is useful, because a natural language is already the basis for requirements specification. To achieve such a specification language based on natural language, Bryant imposes constraints on the natural language that help achieve a sufficient degree of formality. The Two-Level Grammar uses a structured form of natural language to formalize these constraints. Specifying requirements using a specification language based on a natural language rather than a conventional natural language has many benefits. One of which is the degree of structure and formality the specification language achieves without sacrificing the flexibility that is given to the requirements engineer to express the requirements in a language very close to a natural language. This degree of structure and formality reduces ambiguity and provides a mechanism for formal analysis of the requirements.

The brief descriptions of RSL and TLG reveal that specification languages have common characteristics. They are able to:

- **Formalize**: all specification languages add formality to the entities they express or capture
- **Provide Structure**: all specification languages add structure between the entities they express or capture
- **Permit Formal Analysis:** specification languages add formality and structure to entities, thereby permitting formal analysis on these entities.

Since RGML, as defined in Chapter 1, is a formal specification mechanism for characterizing the structure, process flow, and activities inherent to requirement generation processes, RGML needs to embody the characteristics found within specification languages, and in particular the ability to formalize and provide structure.

2.1.3 RGML versus Specification Languages

All the specification languages currently found in the field of requirements engineering focus on specifying individual requirements in a manner that helps eliminate ambiguity which might arise while others interpret these requirements. RGML differs

from these requirements specification languages, because it is designed to specify the **process**, not the individual requirements, by which the requirements engineer gathers requirements. By specifying and formalizing requirements generation processes, RGML helps to eliminate any ambiguity that might arise while requirements engineers execute these processes. In fact, RGML and requirements specification languages complement each other, because requirements specification languages formally specify the requirements while RGML formally specifies the process used to gather these requirements.

In summary, RGML inherits several characteristics from specification languages. However, unlike all the other specification languages, which are designed to specify the individual requirements, RGML is designed to specify the process used to gather requirements, i.e. the requirements generation process. This design feature makes RGML a process description language (PDL). Although there are a number of existing process description languages, none can fulfill the objective that has led to the development of RGML, which is capturing a requirements generation process. The next section concentrates on explaining what process description languages are, identifying what characteristics RGML inherits from these languages, and highlighting how RGML differs from these existing process description languages.

2.2 Process Description languages

This section provides the definition of process description languages and the details of two of these languages that are based on the Extensible Markup Language (XML). The intent is to identify some of these languages' characteristics which RGML shares. The section concludes with a discussion of the similarities and differences between RGML and other PDLs.

2.2.1 Definition of a Process Description language

RGML is a type of PDL. A PDL is a type of specification language designed to formally capture the activities, steps and flow of a process. PDLs differ depending on the type of process the PDL is designed to capture. In general, there are two types of PDLs.

One type is specialized in that the PDL is designed to capture only a certain type of process. The other type is more generic in that the PDL is designed to capture a wider variety of processes, which are not specific to any particular domain. Specialized PDLs are usually equipped with elements and constructs which are relevant only to a particular type of process and which aid in the description of the activities, steps, and the flow of execution within that particular type of process. Generic PDLs should be flexible enough to be decoupled from any single process type and expressive enough to be able to represent fundamental process information for multiple types of processes.

Both types of PDLs can be defined using various kinds of notations. Some use algebraic notations, others use formal languages and still others use only diagrams. However, with the development and maturation of XML [Bray, 1998], there has been a shift towards using XML to define new PDLs because of its modularity and "tag-centric" syntax, which make it a natural fit for representing ordered sequences and hierarchies.

The design of a generic PDL based on XML depends on how the designer characterizes the process. Usually the designer reflects this characterization in a set of primitive constructs the language uses to describe a process. The designs of specialized PDLs are very similar to generic PDLs except that the specialized PDLs have a wider set of constructs, which is a result of adding other constructs related to a particular domain to the primitive set of constructs. Hence, to identify the different characteristics of PDLs, the next two sections provide a discussion of two particular process description languages.

2.2.2. Two Process Description Languages

Since one of the PDLs is generic and the other specialized, the two languages are different. The reason for the discussion of these two different types of PDLs is to ascertain how the basic set of primitive constructs of a PDL depends on the type of process the designer intends to have the PDL describe.

The first of the two languages is the Process Specification Language (PSL), a **generic** process description language. There are two versions of PSL, one based on a

lexicon (a set of symbols) and a grammar (a specification of how these symbols can be combined to make well-formed formulas) [Schlenoff 2000], and the other on XML [Schlenoff 1997]. The latter, named PSL-XML, is of particular interest, because both PSL-XML and RGML are based on XML notation. PSL-XML has a basic set of primitive constructs which comes from the core PSL ontology [Schlenoff 1998]. This set of primitive constructs consists of:

- **Activity:** A class or type of action. For example, painting a house is an activity. It is the class of actions in which houses are being painted.
- **Activity Occurrence:** An event or action that occurs at a specific place and time, or, in other words, an instance or occurrence of an activity. For example, painting John's house in Baltimore, Maryland at 2 PM on May 25, 2001 is an occurrence of the "paint house" activity .
- **Time Point:** An instant separating two states. For example, the point at which the first coat of paint is dry, but before the point at which the second coat has been started is the time point.
- **Object:** Anything that is not a time point or an activity. For example, John's house is an object.

PSL-XML uses these four primitive constructs to describe any process. It embeds XML tags representing these four constructs within each other to describe the structure of the process and the hierarchy of the activities. While PSL-XML captures a process, it references any object used within the process, but does not define it. Another language, Resource Description Framework (RDF) [Lassila 1999], is responsible for defining all the objects referenced in the process. Example 3 shows how PSL-XML expresses the activity of preparing a meal of leftovers. The activity has two sub-activities: reheating the leftover food and reading a newspaper (while waiting for the food to be ready).

```
<activity id="a4">
      <name>prepare leftovers</name>
      <activity id="a5">
            <name>reheat food</name>
```

```
            <resource rdf:resource="leftovers.rdf#Oven"/>
            <resource rdf:resource="leftovers.rdf#FrozenMeal"/>
      </activity>
      <activity id="a6">
            <name> read newspaper</name>
            <resource rdf:resource="leftovers.rdf#Newspaper"/>
      </activity>
</activity>
```

Example 3. PSL-XML representing a process

As Example 3 illustrates, PSL-XML uses tags to represent activities. Sub-activities are embedded within their parent activities. All the objects in this example, such as `Oven`, `FrozenMeal` and `Newspaper`, are referenced as resources and are described in other RDF files.

The second of the two languages is the XML Processing Description Language (XPDL), a **specialized** process description language. XPDL focuses only on the description of business workflows, which are different from generic processes. Workflows are the flow of business relevant knowledge encoded in business documents within and across an organization's boundary [Karakostas 2002]. Workflows also include the routing of tasks from person to person in sequence, thereby allowing each to make a contribution before moving to the next stage. XPDL's basic set of primitive constructs reflects this particular type of process. All the primitive constructs in XPDL are targeted for expressing the description of workflows. Since the XPDL Specification [WFMC 2002] provides the details of the basic set of primitive constructs and their functionality, only a list of the set's constructs follows here:

- Package
- Workflow Process
- Activity
- Transition
- Application
- Data Field (Workflow relevant data)

- Participant

Since an example of a meaningful process expressed by XPDL is lengthy due to the number of primitive constructs in XPDL, Example 4 shows only the syntax of two constructs in XPDL, the activity and the datafield, and how they are expressed using XML:

```
<Activity Id= "Supplier_build" Name = "Supplier_build">
      <Description>The build process is run at supplier</Description>
      <Implementation><No/></Implementation>
      <Performer>Supplier</Performer>
      <StartMode><Manual/></StartMode>
      <FinishMode><Manual/></FinishMode>
<Activity/>

<DataField Id= "Site_OK" Name= "Site OK">
      <DataType>
            <PlainType Type= "BOOLEAN" />
      </DataType>
</DataField>
```

Example 4. Activity and Datafield primitive constructs in XPDL

XPDL and PSL, and in fact most XML-based PDLs, are similar in their format and structure, except for their set of primitive constructs. Since XPDL is a specialized PDL, its set of primitive constructs is bigger and more focused toward the type of process it intends to describe, in this case workflows, than a set in a generic PDL like PSL. Even if two languages have the same primitive construct, like the activity primitive in PSL and XPDL, each language can describe the primitive construct in a different way. The syntax of each primitive construct differs, based on its definition in the language.

Regardless of the type of PDL, every XML-based PDL should have a basic set of primitive constructs that enables the language to:

- Describe the **organization and hierarchy** of the activities within the process.
- Describe the **steps** that need to be executed in order to finish the process.
- Describe the **activities** within a process. If the PDL itself does not describe the details of the activities, then at least it needs to support another language which can describe the activities.

RGML needs to embody all three of these characteristics to achieve its goal of capturing a requirements generation process. Therefore, for RGML to reflect these characteristics, it has to have a basic set of primitive constructs that empowers it to describe the structures, steps, and activities of a process. Chapters 3, 4 and 5 provide detailed discussions of this basic set of primitive constructs, but first it is important to highlight some similarities and differences between RGML and these other process description languages.

2.2.3 RGML and XML-Based Process Description languages

Since RGML is designed to capture the activities, steps, and flow of a requirements generation process, it is a process description language. Based on the definition in Section 2.2.1 of this chapter, RGML is classified as a specialized process description language, because it focuses on capturing a specific type of process - the requirements generation process. Hence, RGML is equipped with a specialized set of primitive constructs for capturing the specifics of requirements generation processes, as well as another set of basic primitive constructs, which enables the language to describe the structure, steps, and activities within a process. RGML is defined using the XML notation, which the next section concentrates on describing in more detail. Consequently, the design of RGML is similar to many of the current XML-based process description languages.

Since RGML, XPDL and PSL belong to the same family, XML-based process description languages, they have many common characteristics, as well as some fundamental differences. One of those differences is that RGML is a specialized PDL, like XPDL, while PSL is a generic PDL. RGML and XPDL, however, do not describe the

same type of processes; RGML describes requirements generation processes, while XPDL describes business workflows. This different focus implies that the sets of primitive constructs of the three languages are different. RGML has a set of primitive constructs which are specialized to capture requirements generation processes; XPDL has a set of primitive constructs tailored to the description of business workflows; PSL has a more generic set of primitive constructs that can be used to describe a wide variety of processes. The generic set of primitive constructs PSL and other generic languages have is neither flexible nor expressive enough to capture the specifics of a requirements generation process. RGML and PSL also differ, because PSL uses another language, RDF, to describe the objects referenced within its process. In contrast, in RGML, everything needed to describe the requirements generation process is defined within RGML. Other than the differences in their primitive constructs, the languages are similar in their characteristics, in their use of tags, and in the way they embed tags to describe the structure and hierarchy of activities.

Since the use of tags in markup languages influences all XML-based PDLs, it is important to understand the influence markup languages and their characteristics have on the design of RGML. The next section, therefore, provides a discussion of the concepts behind markup languages and the resemblance between markup languages and RGML.

2.3 Markup Languages

This section provides the definition of markup languages and an explanation of the role and definition of XML in markup languages. It also concentrates on presenting the concepts and characteristics of markup languages and relating them to RGML.

2.3.1 Definition of Markup Language.

Historically, the term “markup” is used to refer to the process of marking the manuscript copy for typesetting with directions for use of type fonts and sizes, spacing, indentation, and so forth. In the computational world, markup refers to the sequence of

characters or other symbols that are inserted at certain places in a text file to indicate how the file should look when it is printed or displayed or to describe the document's logical structure. The markup indicators are often called "tags." Hence markup languages are languages that use tags to describe certain aspects of the marked items. There are two types of markup languages:

- **Procedural Markup Language**: In this type of language the markup is concerned with the appearance of the text - its font, spacing, and so forth.
- **Descriptive or Declarative Markup Language**: In this type of language, the markup is concerned with the structure or function of the tagged item.

When a designer uses a markup language, the product is a document containing a number of tags surrounding certain information. These tags give meaning (either procedural or descriptive) to the data within them. A simple example from a very famous procedural markup language, Hyper Text Markup Language (HTML) [Berners-Lee 1995], is:

```
<b> My name is Ahmed </b>
```

In this example, opening and closing `<b>` tags surround the piece of data, which is “My name is Ahmed.” These tags indicate that when the piece of data is displayed, it should be in **bold**. Changing the markup tag for the same piece of data gives a new meaning to how the piece of data should be displayed. Another simple HTML example is:

```
<i> My name is Ahmed </i>
```

In this example, the tags indicate that when the piece of data is displayed, it should be in *italics*. Changing a tag results in a change in the way the piece of data is displayed.

In descriptive markup languages, which are more relevant to RGML than procedural markup languages, the tag does not govern the *display* of a piece of data; instead it governs the *meaning* of the piece of data. An example of a descriptive markup language is:

```
<Book> Ahmed's Life </Book>
```

In this example, the `<Book>` tag indicates that "Ahmed's Life" is a book. If the tag is changed then the meaning associated with the piece of data changes, as, for example:

```
<Video> Ahmed's Life </Video>
```

In this example, the `<Video>` tag gives a new meaning to the piece of data, and that is that "Ahmed's Life" is a video

Any markup language is based originally on the Standard Generalized Markup Language (SGML) [ISO 1986], a standard for how to specify a document markup language or tag set. SGML is not in itself a markup language, but a description of how to specify one; it is a "metalanguage," or a language for describing other languages. SGML is large, powerful, and complex. As a lightweight cut-down version of SGML, XML keeps enough of SGML's functionality to make it useful, but removes all the optional features which make SGML too complex. XML is also a "metalanguage," which allows the design of customized markup languages for a variety of purposes. The designers of most of the new markup languages choose XML, instead of SGML, to define their language because of the simplicity of XML. After a markup language is defined, it has a set number of elements and attributes that it uses.

In accordance with the definitions given above, RGML is considered a declarative markup language. However, Section 2.2 in this Chapter defines RGML as a process description language. In fact, RGML is a process description language in the form of a markup language. Since the mechanism used to define a process description language in

the form of a markup language is XML, the designer of RGML uses XML to define a new markup language (RGML) that consists of elements and attributes that are used to describe a requirements generation process.

Therefore, RGML is considered:

- **A Process Description Language**: because it is used for the purpose of describing requirements generation processes.
- **A Markup Language**: because it consists of a set of tags that describe the structure or function of items.
- **A Scion of XML**: because as a new markup language, RGML is defined using the "metalanguage" XML.

Consequently, the newly designed RGML has an unusual range of characteristics and flexibility that makes it ideal for the challenge of describing any requirements generation process. RGML provides structure, formalism, organization, and hierarchy of the activities of the requirements generation process. At the same time, RGML captures the steps in the process that provide guidance to the requirements engineer as he or she is executing the requirements generation process.

3. Overview of RGML

The design process for RGML requires the consideration of many alternatives in order to find the model best suited to the RGML's objective of characterizing and capturing generation processes. This chapter provides an analysis of a requirements generation process to identify the characteristics and properties that dictate the design of RGML.

3.1 Analysis of a Process

A list of the published requirements generation processes, which several organizations use, include the Requirements Generation Model (RGM) [Arthur 1999], the Agile Requirements Model [Ambler 2001], the Requirements Triage [Davis 1999], the Knowledge Level Process Model [Herlea 1999], and Volere [Robertson 1999]. Careful consideration of all these processes has led to the selection of the RGM as a representative process because of the variety of different characteristics and properties it contains. This variety is necessary for the design of RGML, because, as a language aims to capture requirements generation processes, the characteristics and properties it processes play a fundamental role in the design of the language. Figure 1 provides an overview of the RGM process framework that highlights the simplicity and richness of the requirements generation process, as well as its variety of tools, techniques, and activities. A brief description of the properties and capabilities of the RGM and a discussion of their impact on the design of RGML follows.

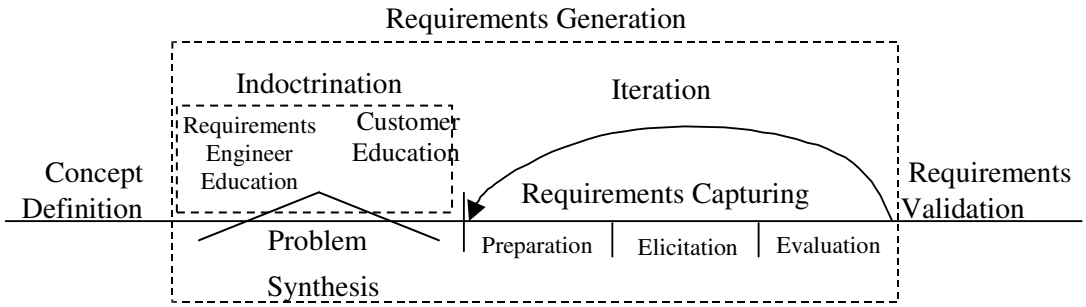

Figure 1. Overview of the RGM Process Framework

As Figure 1 shows the requirements generation process of the RGM consists of two phases (parts) *Indoctrination* and *Requirements Capturing*. Both are necessary for the process to achieve its goal of guiding the requirements engineer through a series of activities to produce a set of "good" requirements. The RGM [Arthur 1999] defines a framework that divides the requirements generation process into three main phases: an initial *Indoctrination* phase, a *Problem Synthesis* phase fused within the *Indoctrination* phase, and an iterative *Requirements Capturing* phase. The *Indoctrination* phase has two parts. In the first part, which takes place before the *Problem Synthesis* phase, the requirements engineer begins his or her education with an overview of the customer's problem domain and needs and a description of the participants' tasks and responsibilities in the requirements definition process. In the second part, the requirements engineer provides the customer with an introduction to the requirements definition process.

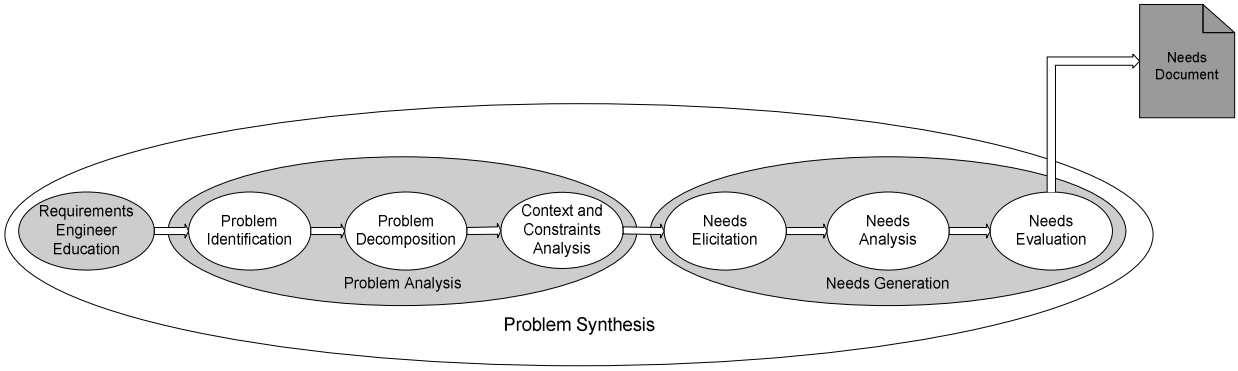

Figure 2. Problem Synthesis phase of RGM

From this overview the requirements engineer is able to move into the *Problem Synthesis* phase, which Figure 2 shows in detail. In this phase, the requirements engineer begins the process of understanding the real problem, as well as assessing and documenting user needs. The objective of the *Problem Synthesis* phase is to guide the requirements engineer to identify and gain a deeper understanding of the problem, the context, and the environment before he or she elicits the actual requirements. This phase consists of two main sub-phases: *Problem Analysis* and *Needs Generation.*

In the *Problem Analysis* phase, the requirements engineer identifies and investigates the root cause of the problem by analyzing and decomposing it into its constituents. During the *Needs Generation* sub-phase, the requirements engineer elicits, documents, and analyzes the needs of the users for each of the problem constituents identified in the *Problem Analysis* phase. These needs are subsequently used for eliciting requirements during the *Requirements Capturing* phase, which is the next phase (see Figure 1).

The objectives of this third phase are directly related to its three sub-phases: *Preparation*, *Elicitation*, and *Evaluation*. Their objectives are respectively:

- to define the scope of the elicitation meeting and ensure that all participants have completed their pre-meeting assignments,
- to enable the requirements engineer to accurately identify and record software requirements as expressed by the customer, and
- to evaluate the contributions of the preceding elicitation meetings, identify unresolved (or new) issues, and determine if an additional refinement iteration is needed.

The RGM also includes a monitoring methodology which operates throughout the requirements generation phase and in tandem with attendant activities to ensure that the requirements engineer follows the proper procedures and protocols. Working together, the framework and monitoring methodology

- add structure and control to the requirements generation activities,
- support the accurate capturing of requirements, and
- promote an effective requirements generation process.

Although there are more details and features of the RGM, this overview clearly shows its flexibility, richness, and capabilities. However, this overview is not sufficient to show how the characteristics and features of requirements generation processes, as

depicted in the RGM, influence the design of RGML. Within a requirements generation process, the main influences on the design of RGML are:

- the concept of structure
- the concept of activities

The concept of structure of a process refers to the arrangement of the different activities within the process, whereas the concept of activities in a process refers to the specifics of each of the activities within a requirements generation process. Since these two concepts influence the design of RGML the most, the next two sections concentrate on explaining their impact and uses RGM to illustrate it.

3.2. The Concept of Structure in Requirement Generation Processes

Understanding the concept of structure within a process requires an understanding of how the structure corresponds to the flow of execution of the process. The structure of a process can be conceptualized as the arrangement of the activities within the process. An examination of the details of the representative requirements generation process, as seen in the RGM, highlights various interesting structures of activities. For illustrative purposes this section presents the different structures found within the RGM in the order of increasing complexity of the structure, not in the order of the phases within the process. The *"Context and Constraints Analysis"* phase located within the *Problem Synthesis* sub-phase (Figure 3), is the first structure of interest.

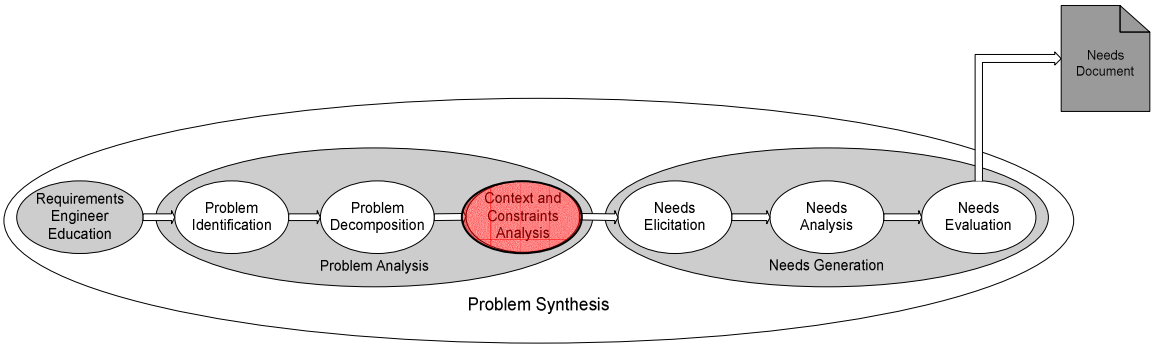

Figure 3. The Context and Constraints Analysis activity within the Problem Synthesis phase.

3.2.1 The Sequence Structure

As Figure 4 shows, three main activities occur within the *"Context and Constraints Analysis"* phase:

- **Identify System Role:** to define system boundaries
- **Identify Actors:** to gather information related to the context of the system
- **Elicit Constraints:** to elicit the constraints for the proposed system.

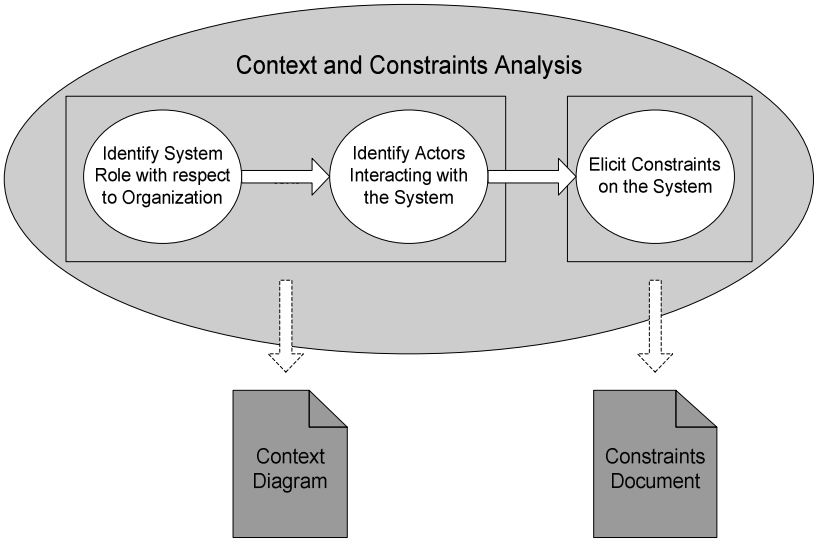

Figure 4. The Context and Constraints Analysis phase.

The *"Context and Constraints Analysis"* phase starts once the requirements engineer gains an in-depth understanding of the problem (of the current system) from the preceding *Problem Decomposition* phase. The requirements engineer uses the *"Context and Constraints Analysis"* phase to develop a scope or boundary within which the system will operate. Since the boundary separates the system from other systems and the real world, it helps identify the inputs the system is processing, as well as the corresponding outputs. To establish the context within which the system must operate, the requirements engineer determines the data exchanged between the proposed system and other systems. Next, he or she identifies the users or actors interacting with the system. Once the requirements engineer identifies the problem and establishes the solution boundaries

(context), it is important that he or she elicits the constraints within which the proposed system must operate through continuous interaction and verification with stakeholders.

Within the structure of the *"Context and Constraints Analysis"* phase lie activities which are arranged in a single sequential order. This arrangement of activities is common in requirements generation processes and, therefore, for a language to have the ability to express requirements generation processes, it must have the ability to capture activities in a sequential structure.

3.2.2 The Iteration Structure

The iteration structure is similar to the sequential structure, except that in the iteration the last activity in the structure loops back to the first activity until a condition is met. The *Problem Decomposition* phase, located within the *Problem Synthesis phase*, as seen in Figure 5, contains an iteration structure.

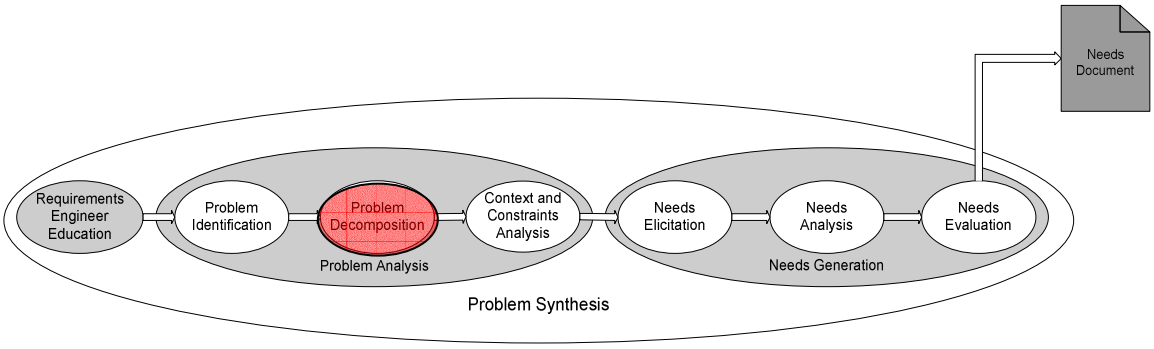

Figure 5. The Problem Decomposition activity within the Problem Synthesis phase.

Figure 6 shows the three main activities found in the *Problem Decomposition* phase:

- Meeting with Stakeholders
- Identifying Key Problem Areas
- Resolving Unaddressed Issues

Although the requirements engineer identifies the actual problem in the current system in the *Problem Analysis* phase, in order to gain additional insights into the problem and, as a result, a better understanding of the customer's needs, he or she uses the *Problem Decomposition* phase to break the problem down into smaller distinct elements. It is then possible to refine their individual characteristics. *Problem Decomposition*, therefore, consists of a series of activities that the requirements engineer executes to obtain problem elements, which aid him or her in pinpointing the customer's needs. *Problem Decomposition* is an iterative process composed of (numerous) structured meetings with stakeholders. To support *Problem Decomposition*, each meeting must have a focused objective and employ structured activities that support the achievement of that objective.

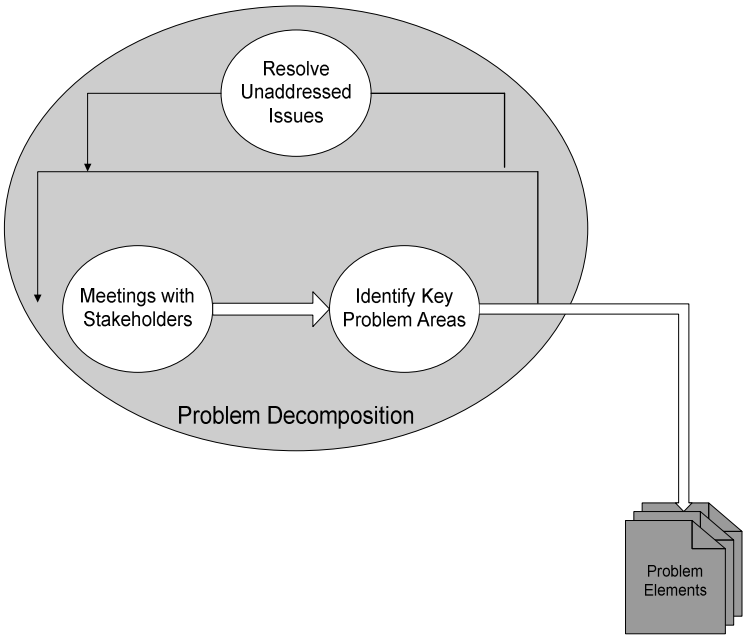

Figure 6. The Problem Decomposition phase

Figure 6, which shows the activities arranged in an iterative order, also shows *"Resolve Unaddressed Issues"* as an optional activity in the *Problem Decomposition* phase. The side concept introduced by the arrangement of the *"Resolve unaddressed issues"* activity is having optional activities within the structure of the process. For

RGML's design to be complete, it must have the ability to capture activities in iterations, and it should also support the concept of optional activities within structures.

3.2.3 The Conditional Split Structure

The second part of the requirements generation process, the *Requirements Capturing* phase (see Figure 1), contains the *Local Analysis* sub-phase. Figure 7, which provides a more detailed diagram of the *Requirements Capturing* phase than Figure 1, shows *Local Analysis* as a sub-phase. As the term 'Local' implies, the requirements engineer conducts the activities of the phase on the set of requirements generated from the elicitation meetings held before this phase. During *Local Analysis*, he or she associates attributes with every requirement. Although not required to associate all attributes with every requirement, the more attributes associated with a requirement, the more accurate the analysis can be.

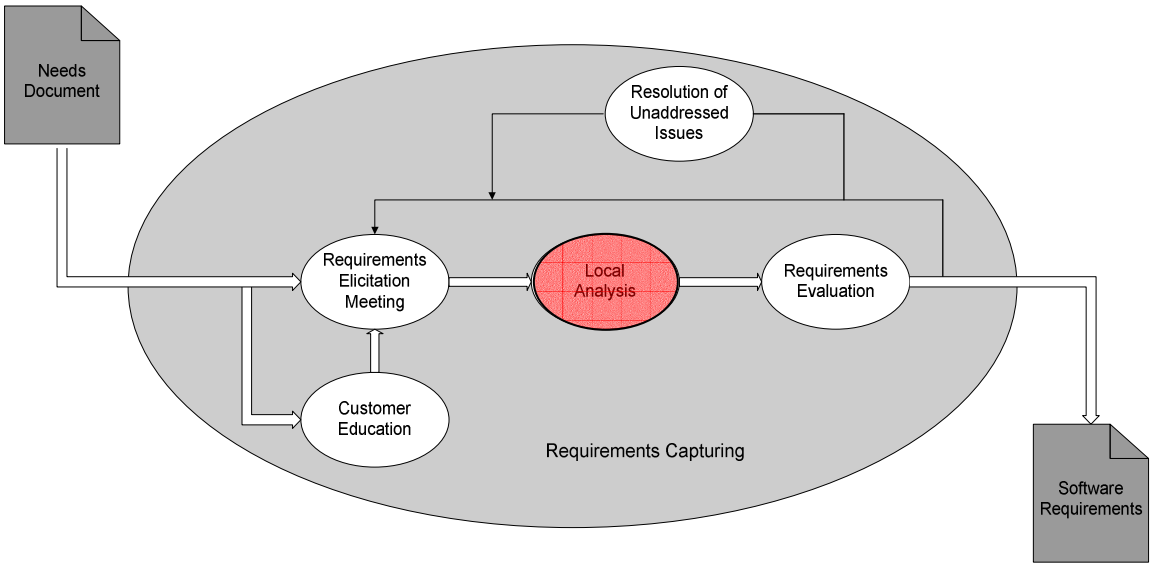

Figure 7. Requirements Capturing Phase with emphasis on Local Analysis

Figure 8 shows the activities within the *Local Analysis* phase, which represents an independent approach to the analysis of software requirements. The group of activities in Figure 8 are arranged in an iterative order. The requirements engineer carries out each of these activities on the list of requirements produced during the requirements elicitation

meeting. However, he or she does not have to execute all these activities, but can choose which activity to execute or ignore. The set of activities that are in a branched iterative order are:

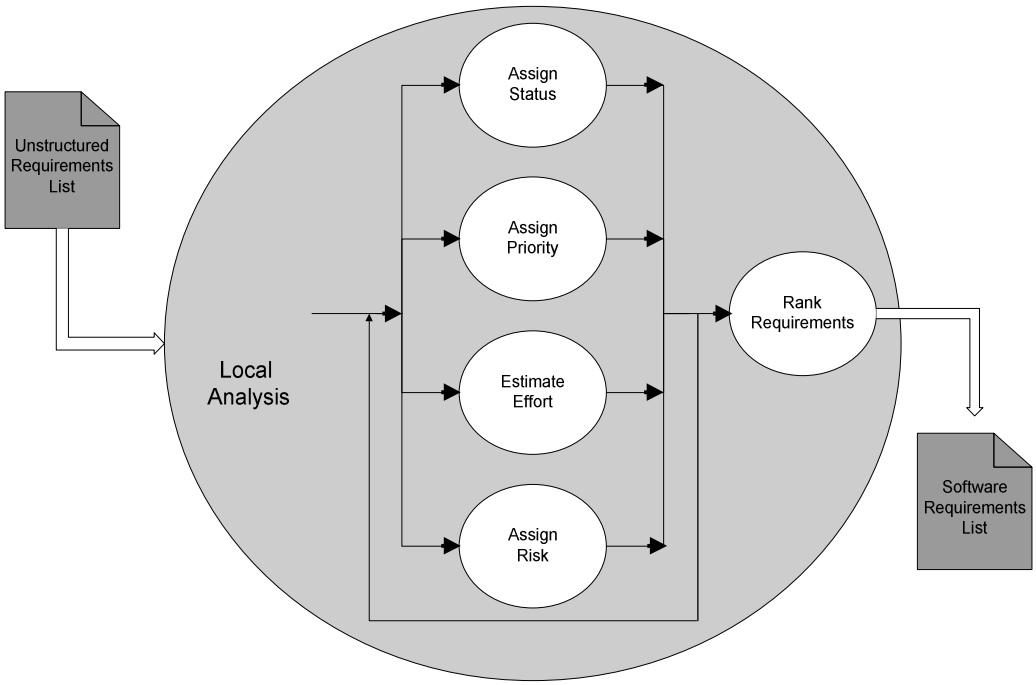

Figure 8. Local Analysis Phase

- **Assign Status:** A status is attributed to each the requirement;
- **Assign Priority:** A priority is attributed to the requirement;
- **Estimate Effort:** The requirements engineer, with the help of the implementation team, estimates the effort required to implement the requirement;
- **Assign Risk:** The requirements engineer assigns a risk attribute to the requirement.

When all the iterations are complete, the *Rank Requirements* activity occurs.

- **Rank Requirements:** During this activity, the requirements engineer ranks the requirements according to the values of the attributes associated with each requirement.

The *Local Analysis* phase embodies the split, a structure different from those presented so far. The split structure can occur in two forms - the conditional and the unconditional. The conditional split requires certain conditions, associated with each branch, to be fulfilled before the execution of that branch can occur. The unconditional split, as depicted by the *Local Analysis* phase, allows the requirements engineer to execute any branch of the split without any conditions. For a language to have the ability to capture the various structures found within requirements generation processes, it must have the ability to capture both types of splits.

The representative sample of structures presented in this section makes it possible to conclude that activities can be arranged in many different orders. Some activities are arranged in an iterative order. Others are arranged in a split order that can be either conditional or unconditional. Many other activities are organized in a simple sequential order. Since activities can assume a wide variety of arrangements, and since the aim of RGML is to capture any requirements generation process, it must have the ability to capture the different arrangements of activities within the process. RGML accommodates the concept of structure in requirements generation processes by having a set of basic structure constructs, such as sequence, iteration, and split. The user of RGML utilizes these structural constructs together with the composite construct (see section 4.2.5) to express the structure of requirements generation processes.

3.3. The Concept of Activities in Requirement Generation Processes

In this section, the focus shifts from the concept of structures in requirements generation processes and its influence on the design of RGML to the concept of activities and its influence on the design of RGML. The concepts differ in that the structure of a process refers to the arrangement of the different activities within the process, whereas the concept of activities in a process refers to the specifics of each of the activities within a requirements generation process. Since the specifics of an activity differ from each other, the challenge in designing RGML is to devise a mechanism that enables RGML to

describe the specifics of any activity. The design of RGML must, therefore, base the mechanism on capturing the specifics of the activity through a number of aspects that all activities have in common.

The specifics of the *"Requirements Elicitation Meeting"* activity of the RGM are used here as our representative example. Figure 9, which gives a more detailed representation of the *Requirements Capturing* phase than Figure 1, highlights this activity.

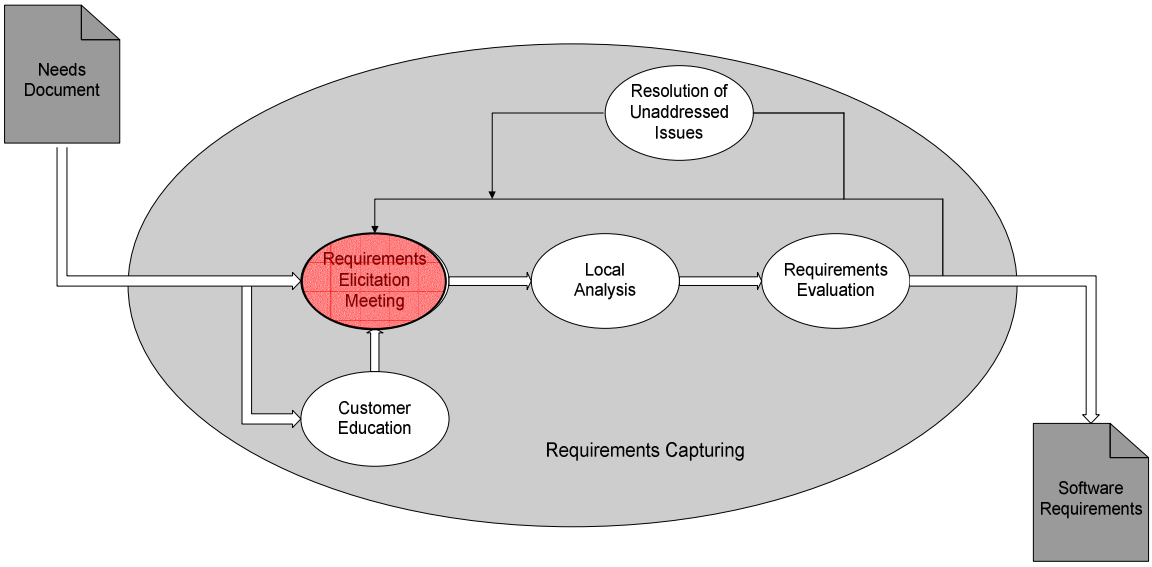

Figure 9. Requirements CapturingPhase with emphasis on Requirements Elicitation Meeting

The requirements engineer conducts the *"Requirements Elicitation Meeting"* activity during the *Requirements Capturing* phase. Before identifying and explaining the specifics of any activity, it is important to know the initial factors that affect the activity. The requirements engineer must take into account any pre-existing condition, also known as preconditions, and fulfill them first. The requirements elicitation meetings cannot begin until the requirements engineer completes the *Problem Synthesis* phase. The completion of the *Problem Synthesis* phase is, therefore, a precondition of the *"Requirements Elicitation Meeting"* activity.

The other initial factor to consider, before explaining the specifics of any activity, is an identification of the objective of an activity in order to conduct the activity in a way that achieves its objective. The objective of the *"Requirements Elicitation Meeting"* activity is to acquire information from stakeholders. With this information, the requirements engineer can identify and analyze the requirements of the new system.

To achieve the objective of any activity, the requirements engineer needs to perform a series of steps that lead to the desired objective. The first step for the requirements engineer in *"Requirements Elicitation Meeting"* activity is to decide on the methodology for the upcoming meeting. Such methodologies include the Joint Application Design (JAD) and the Participatory Design (PD). JAD and PD differ in the selection process of meeting participants. The former advocates participant representation from an entire cross-section of the hierarchy of the organization, while the latter advocates selection of participants from the same hierarchical level. When the requirements engineer chooses the methodology and identifies the participants, the participants receive a notification outlining the agenda of the forthcoming meeting. The requirements engineer assigns definitive tasks to all or particular participants to complete in preparation for the meeting. For example, the requirements engineer may assign some stakeholders the task of presenting information regarding a viable approach or an idea to the other participants. An experienced requirements engineer can define templates that less experienced requirements engineers can use to prepare meeting agendas.

When, the requirements engineer has the agenda ready he or she dispatches the meeting notification to all stakeholders. He or she conducts the meeting according to the guidelines of either the JAD or the PD approach, depending on which one is selected. The requirements engineer must be sure that the proceedings of the meeting are documented, because they are an important source of requirements, albeit an unstructured one. After the meeting, the requirements engineer organizes the minutes and pertinent documents and circulates them among all meeting participants and relevant stakeholders in order to elicit feedback about the correctness of collected information.

3.3.1 Activity Preconditions

From this brief description of the specifics of the *"Requirements Elicitation Meeting"* activity, it is possible to point out some of the common aspects, which challenge the designer and influence the design of RGML. First, before going into the specifics of the activity, it is necessary for the designer of the requirements generation process to state the preconditions of that activity. Preconditions are a common aspect of processes, because usually in processes, activities are built upon the results of other activities. Therefore, it is important to use preconditions to prevent the execution of one activity until after the execution of another activity. Hence any language, including RGML, intending to capture the details of activities must support the notion of preconditions while capturing activities. While capturing the details of activities RGML must support the fact that each activity has an objective. The more complex issues of activities show that each activity has a series of steps that the requirements engineer must carry out to fulfill the objective of the activity. From the previous description of the *"Requirements Elicitation Meeting"* activity it is possible to identify five main steps:

- Identify primary stakeholders
- Set meeting agenda
- Dispatch meeting notification
- Conduct the meeting
- Record minutes and document proceedings

3.3.2 Activity Artifacts and Actions

A closer look reveals differences between some of the previous steps. Some steps, such as *"Set meeting agenda"* and *"Record minutes and document proceedings"* deal with documents. *"Set Meeting Agenda"* produces documents according to pre-defined templates. *"Record minutes and document proceedings"* step produces documents according to certain standards and templates. Other steps, such as *"Identify primary*

stakeholders" and *"Dispatch meeting notification,"* deal with applications. The requirements engineer might can use tools like the "Hierarchy Generator" to complete the *"Identify primary stakeholders"* step. The *"Dispatch meeting notification"* step employs an e-mail application to send the notification to the participants. It is likely that steps associated with applications will need to instantiate the applications or tools associated with the step to automate and facilitate the process. A different and common kind of step, such as *"Conduct the meeting,"* does not deal with anything in the computational arena, but just advises and guides the requirements engineer on how to carry out an activity in the real world.

Therefore, in addition to having the ability to capture the different steps of activities, RGML must have the ability to capture the individual features of these steps, such as artifact production (any documents the process produces or consumes) and application instantiation.

3.3.3 Activity Exceptions

The *"Requirements Elicitation Meeting"* activity calls attention to "activity exceptions," another aspect of any activity. The requirements engineer determines which activity alternative to execute by assigning each alternative a set of conditions. For example, in the *"Requirements Elicitation Meeting"* activity, choosing a methodology for the upcoming meeting determines which activity alternative to execute. The alternative means that the details of all five steps depend on the methodology the requirements engineer chooses. When the requirements engineer chooses the JAD, he or she is opting for a specific way to identify the primary stakeholders that differs from the PD. Since the selected approach applies to all succeeding steps, "activity exceptions" are required to implement conditional execution of steps within an activity. Due to the importance and frequent existence of "activity exceptions" in activities of requirements generation processes, RGML needs to have the ability to capture the "activity exceptions" as part of its capabilities. Therefore, the four main aspects of activities that RGML should have the ability to express are:

- Activity precondition(s)
- Activity objectives
- Steps within an activity (with the different features accompanying each step)
- Activity exceptions.

3.4 Structure of the Language

Sections 3.2 and 3.3 illustrate how the concepts of structure and activity in requirements generation processes are translated into two main components of RGML. The first component of RGML expresses the structure of processes using four main structural constructs; *sequence*, *iteration*, *split* and *composite* (explained in section 4.1.5). The second component of RGML describes activities by capturing preconditions, objective, steps, and exceptions of an activity. In addition to the two main components of structure and activity, RGML includes a third component, *definitions*, which does not deal with describing anything about the actual process. Since the requirements engineer uses the definitions component as a place in the language to define artifacts, templates, actions, and structures, he or she needs to define these elements within the definitions component of the language before he or she can reference them in the structure or activity components of RGML.

Since RGML is a markup language, it uses tags to express different elements. The three main elements within RGML correspond to the three main components of the language: *ProcessStructure*, *ActivityDetails*, and *Definitions*. To introduce the actual syntax of RGML, Example 5 presents a high level abstract of the language:

```
<RGML>
      <ProcessStructure>
            The flow of control of the activities is nested here
      </ProcessStructure>
      <ActivityDetails>
            The description of the activities is nested here
      </ActivityDetails >
      <Definitions>
            All the templates, artifacts, and types are nested here
      </Definitions>
   </RGML>
```

Example 5. Framework of RGML Syntax

This example illustrates the framework of RGML. The language starts with an `<RGML>` tag under which everything else is nested. Nested within the `<ProcessStructure>` tag are all the structure oriented constructs, which include the `<Sequence>`, `<Iteration>`, `<Split>`, and `<Composite>` tags. All the activities in the process are described under the `<ActivityDetails>` tag. All the elements that need to be defined are nested under the `<Definitions>` tag. Chapter 4 provides a detailed explanation of each of the three main components of RGML, `<ProcessStructure>` `<ActivityDetails>` and `<Definitions>`.

For a better appreciation of the language, two examples codified in RGML follow. Example 6 shows how RGML expresses the structure of the local analysis phase (see Figure 8, Section 3.2), and Example 7 shows how RGML expresses the details of the *"Requirements Elicitation Meeting"* activity (see Section 3.3). These examples are not expressed using the full syntax of RGML; for illustrative purposes some detail tags and syntax are not included.

Figure 10 re-visits the structure of the *Local Analysis* phase; Example 6 shows RGML code representing the structure of the *Local Analysis* phase. As Figure 10 shows, the *Local Analysis* phase consists of an iteration (blue lines) of an unconditional split (red

lines) between four activities (*Assign Status, Assign Priority, Estimate Effort,* and *Assign Risk*). Following the iteration is the single activity *Rank Requirements* (green line). RGML places single activities within sequence constructs; Section 4.2.2 provides a detailed explanation.

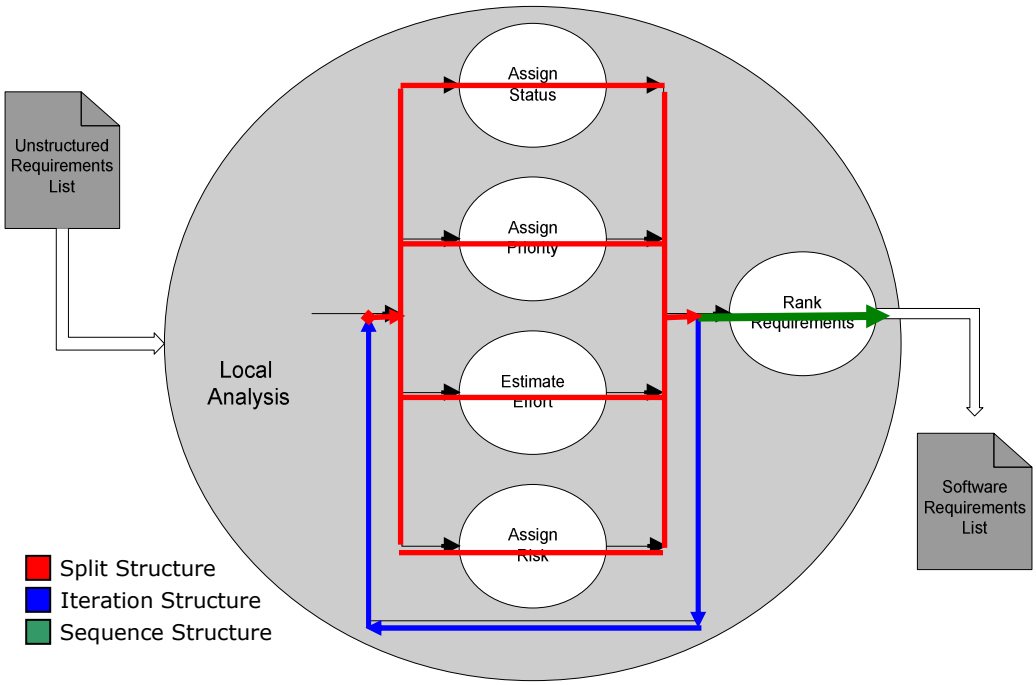

Figure 10. Local Analysis phase with highlights on different structures

```
<ProcessStructure>
  <Iteration>
    <Split>
      <Branch>
         <RefActivity id=”AS”>Assign Status</RefActivity>
      </Branch>
      <Branch>
         <RefActivity id=”AP”>Assign Priority</RefActivity>
      </Branch>
      <Branch>
         <RefActivity id=”EE”>Estimate Effort</RefActivity>
      </Branch>
      <Branch>
         <RefActivity id=”AR”>Assign Risk</RefActivity>
      </Branch>
   </Split>
 </Iteration>
 <Sequence>
      <RefActivity id=”RR”>Rank Requirements</RefActivity>
 </Sequence>
</ProcessStructure>
```

Example 6. RGML code for the structure of the Local Analysis phase

Example 7 presents RGML code that captures the details of the *“Requirements Elicitation Meeting”* activity. Since the syntax associated with activities has not yet been presented, the code concerning the activity exceptions is not presented in this example.

```
<ActivityDetails>
   <Activity>
      <Precondition> successfully completed the needs generation phase
      </Precondition>
      <Goal> identify and capture requirements as communicated by the
      customer and the users </Goal>
      <Flow>
         <Step>
            <Name> Identify primary stakeholders </Name>
```

```
            <Action> <Type> Hierarchy Generator </Type> </Action>
         </Step>
         <Step>
            <Name> Set meeting agenda </Name>
            <Action>
               <Type> Word processor </Type>
               <Artifact> Agenda </Artifact>
            </Action>
         </Step>
         <Step>
            <Name> Dispatch meeting notification</Name>
            <Action> <Type> E-mail Program </Type> </Action>
         </Step>
         <Step>
            <Name> Conduct the meeting</Name>
         </Step>
         <Step>
            <Name> Document minutes and proceedings </Name>
            <Action>
               <Type> Word processor </Type>
               <Artifact> List of Requirements </Artifact>
            </Action>
         </Step>
   </Activity>
</ActivityDetails>
```

Example 7. RGML code capturing the "Requirements Elicitation Meeting" activity

In conclusion, this chapter uses a representative requirements generation process, the RGM, to show how the concepts of process-structure and activities within requirements generation processes influence the design of RGML. Two main components corresponding to the two main concepts of process-structure and activity, along with a third component, the definitions used in the language, comprise the framework of RGML. The next chapter presents the syntax and semantics of the different constructs found within each of RGML's components.

4. Constructs of RGML

The previous chapter provides an overview of the Requirements Generation Markup Language (RGML), its construction and the rational behind having the components of the language. This chapter concentrates on presenting the details and syntax of the different elements and constructs used within each of the *process-structure*, *activities* and *definitions* components of RGML. It provides a discussion of the syntax and the associated semantics of the different constructs used by RGML to define the structure and the activities of requirements generation processes. We also show how to use each of the constructs of the language and provide examples of the different situations where various constructs are used. This chapter is divided into three main sections. Each of these sections, which corresponds to one of the main components in RGML, focuses on describing the constructs and elements of RGML.

4.1 The Process-Structure

The first component of RGML captures the *structure* of the requirement generation process. The structure of a process is the flow of execution of the process, which intuitively is the arrangement of the activities within the process. Using a combination of constructs, where each construct expresses a different possible organization of activities within a process, RGML is able to capture the structure of requirements generation processes. RGML requires these constructs to be embedded between the opening and closing `<ProcessStructure>` tags. While defining the structure of the process, RGML avoids defining any details of individual activities within the process. Instead, RGML's process-structure component is responsible for the description of the arrangement and organization of the activities and the sequence of their execution. Example 8 below shows the position of the *Process-Structure* component relevant to the other components within RGML.

```
<RGML>
      <ProcessStructure>
            The flow of control of the activities is nested here
      </ProcessStructure>
      <ActivityDetails>
            The description of the activities is nested here
      </ActivityDetails >
      <Definitions>
            All the templates, artifacts, and types are nested here
      </Definitions>
</RGML>
```

Example 8. Position of the Process-Structure component relevant to the RGML.

4.1.1 Introduction to Structures

For RGML to have the ability to capture and express requirements generation processes, it needs to have the expressive power to describe a variation of activities found within requirements generation processes. To achieve this, RGML uses four tags; each tag represents a unique structure within the language: `<Sequence>`, `<Iteration>`, `<Split>` and `<Composite>`. When used together, either in a sequential or in an embedded fashion, they provide the flexibility to describe a variety of activity sequences. The designer nests these four tags within the opening and closing `<ProcessStructure>` tags. Each of the four structural tags describes the arrangement of activities within a certain segment of a requirements generation process. Within the structural tags the designer uses the activity's *Id* when he or she references an activity. A description of the activity details is provided later under the `<ActivityDetails>` section of the RGML specification. All activity references are embedded within one of the four basic structure tags, which, in turn, are embedded within the `<ProcessStructure>` tag. In the case of a single activity, the designer embeds the activity within a `<Sequence>` tag (Section 4.1.2).

Figure 11, shows an abstract of a hypothetical requirements generation process.

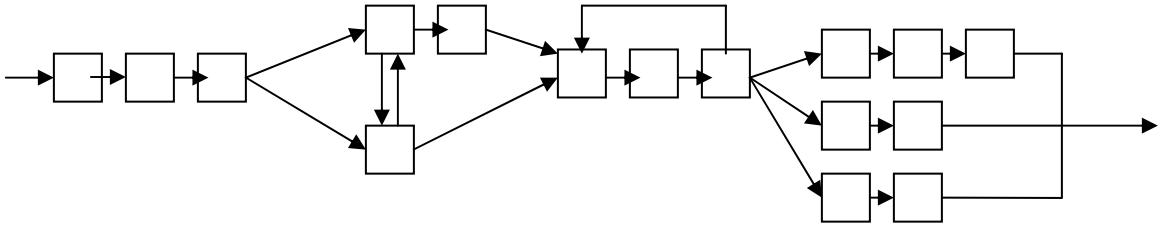

Figure 11. Hypothetical Requirements Generation Process

To express the requirements generation process shown in Figure 11, in RGML, first, the designer examines the process and divides it into segments where one of the three basic structural construct (sequence, iteration or split) can describe the activities within each segment. Any set of activities the designer can not describe using one of the three basic structural constructs, he or she describes using the composite construct (for details see section 4.1.5).

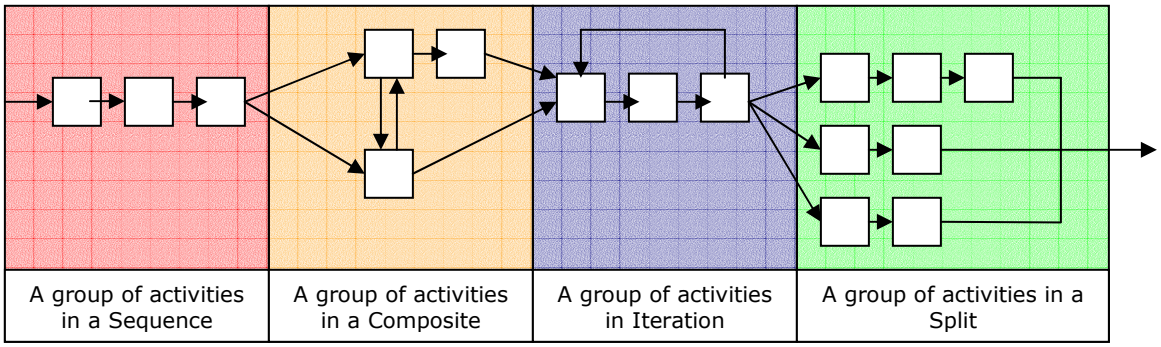

Figure 12. Hypothetical Requirements Generation process divided up into a sequence of segments

Figure 12 illustrates how a designer divides the hypothetical process into segments in which he or she groups together the activities that fit within a common structural construct. As a side note, RGML supports multiple levels of segmentation, which means that the presence of other structural segments within one structural segment of the process is also possible. This phenomenon is common within iterations and splits. As Figure 12 shows, after the designer identifies the segments, he or she expresses each of these segments using one of the structural tags in RGML.

Within each tag, the designer references all the activities found within that structural segment of the process. He or she then describes the details of each of the activities within the process in the `<ActivityDetails>` section of the language. While doing this the designer can reference any auxiliary elements such as artifacts, templates, actions or applications, which the activity produces or consumes. Afterwards, the designer must define in the `<Definitions>` section of the RGML all of the elements (artifacts, actions, templates, or applications) he or she referenced in the `<ActivityDetails>` section. Example 9 illustrates how RGML expresses the hypothetical process given in Figure 11 and Figure 12 and also places the `<ProcessStructure>` section in perspective to other RGML components.

```
<RGML>
   <ProcessStructure>
         <Sequence>
               The activities within this structure are here
         </Sequence>
         <Composite>
               The activities within this structure are here
         </Composite>
         <Iteration>
               The activities within this structure are here
         </Iteration>
         <Split>
               The activities within this structure are here
         </Split>
   </ProcessStructure>
   <ActivityDetails>
         The specifics of all the activities referenced under the
         <ProcessStructure> tag are described here (See Section 4.2)
   </ActivityDetails>
   <Definitions>
         Any elements referenced under the <ActivityDetails> tag are defined
         here (See Section 4.3)
   <Definitions>
```

```
</RGML>
```

Example 9. A framework of RGML expressing a requirements generation process

The next sections explore each of the four structural constructs in detail in order to illustrate their syntax and semantics and any special functionalities associated with the constructs.

4.1.2 Sequence

The first of the structural constructs is the sequence construct, depicted in RGML by the `<Sequence>` tag. The designer uses the sequence construct to capture a group of activities that the requirements engineer executes one after the other, as seen in Figure 13.

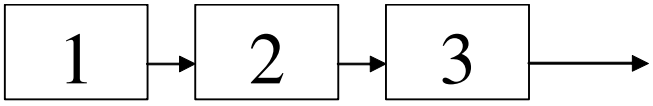

Figure 13. Sequence Flow Diagram

Figure 13 shows a flow diagram of three activities arranged in a sequential arrangement. The designer of a process can express such a structure in RGML as follows:

```
<ProcessStructure>
      :
      <Sequence>
            <RefActivity id=1> Number 1 </RefActivity>
            <RefActivity id=2> Number 2 </RefActivity>
            <RefActivity id=3> Number 3 </RefActivity>
      </Sequence>
      :
<ProcessStructure>
```

Example 10. Sequence Construct in RGML

The sequence construct is essential to the objective of RGML, because it permits one to organize activities in a sequential order in processes.

As mentioned previously, the <ProcessStructure> tag does not directly embed references to single activities; it only embeds one of the four basic structural constructs. Therefore, if the designer has a situation where a single activity precedes a structural segment (see Figure 14), he or she must embed the single activity within a <Sequence> tag.

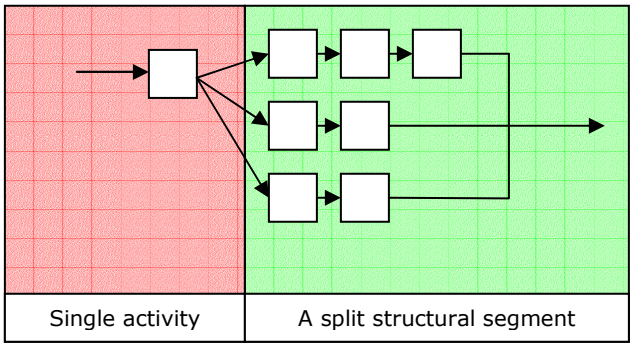

Figure 14. Single Activity Sequence Diagram

Example 11 illustrates the process shown in Figure 14. The main focus of this example is to show how RGML requires that any single activity be incorporated within a <Sequence> tag and embedded within the <ProcessStructure> tag.

```
<ProcessStructure>
      <Sequence>
            <RefActivity id=1> Number 1 </RefActivity>
      </Sequence>
      <Split>
            :
            :
      </Split>
<ProcessStructure>
```

Example 11. Sequence Construct having one Activity

Since RGML allows the designer to directly embed only certain tags within the `<Sequence>` tag, any other tags cause a syntax error. Below is brief description of each of the tags RGML permits the `<Sequence>` tag to nest:

- **The `<IncludeStructure>` tag:** Not all structures in RGML must be defined within the `<ProcessStructure>` tag. The designer can pre-define reusable-structures that are frequently used within the process in the `<Definitions>` section of RGML, and then later reference them in various locations in the process when needed. The designer uses the `<IncludeStructure>` tag to indicate that at that specific position in the process the reusable-structure which the `<IncludeStructure>` tag references should be inserted. Section 4.1.6 provides a detailed explanation of the concept of including reusable-structures.

- **The `<RefActivity>` tag:** This tag, which is to reference an activity, is the most commonly used tag within any structural tag. This tag only uses the *`Id`* of the activity the designer needs to reference at this position. The specifics and details of the activity itself are found under the `<ActivityDetails>` tag (see section 4.2).

- **The `<MilestoneScope>` tag:** A milestone is an element RGML uses to indicate the occurrence of a specific incident. The designer of a requirements generation process can insert and set milestones to either “On” or “Off,” depending on the intended use of the milestone. The designer can also define a milestone to be only operational in certain areas of the process, which is known as the scope of the milestone. The scope of the milestone is limited to whatever is between the opening and closing tags of the `<MilestoneScope>` tag. Chapter 5 Section 1 provides more detailed explanation of the concept of milestones and their scopes.

- **The `<Group>` tag**: RGML uses this tag to group a set of structures together under one name. The designer places the structures he or she wants to group together between the opening and closing `<Group>` tags. Section 4.1.6 gives more details of the concept of grouping structures.

4.1.3 Iteration Construct

The second of the structural constructs is the iteration construct, codified in RGML through the `<Iteration>` tag. The designer uses the iteration construct to capture a group of activities that the requirements engineer executes multiple times; however, after the execution of the last activity in the group, the requirements engineer executes the first activity again until an exit condition is satisfied, as Figure 15 shows. The syntax of the `<Iteration>` tag resembles the syntax of the sequence tag except that it has an additional tag, the `<Exit>` tag, used to capture the exit condition the requirements engineer must satisfy in order to exit the execution of the iteration construct.

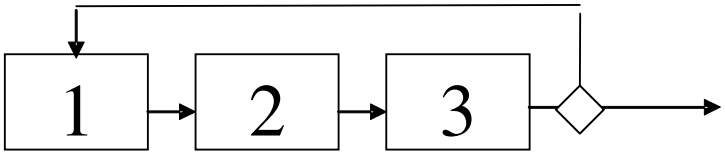

Figure 15. Iteration Flow Diagram

Figure 15 depicts a flow diagram of three activities arranged in a sequential manner. The designer of a process can express such a structure in RGML as follows:

```
<ProcessStructure>
      <Iteration>
            <RefActivity id=1> Number 1 </RefActivity>
            <RefActivity id=2> Number 2 </RefActivity>
            <RefActivity id=3> Number 3 </RefActivity>
            <Exit>
                ***Exit condition is placed here (See Chapter 5 Section 2)***
            </Exit>
      </Iteration>
</ProcessStructure>
```

Example 12. Iteration Construct

Example 12 does not show the details of the exit condition, because the format in which the designer expresses the exit condition has not yet been presented. The designer

expresses the exit condition in a special format that is used throughout RGML to express condition of any type. This special format gives the designer of the process the ability to express the iteration's exit condition(s) using Boolean operators to join an unlimited number of conditions based on questions, artifacts or milestones. Chapter 5 Section 2 provides an explanation of the standard format RGML uses to capture conditions.

As it does with the `<Sequence>` tag, RGML allows the designer to embed only certain tags directly within the `<Iteration>` tag. Any other tags will cause a syntax error. In addition to those tags that can be embedded within the `<Sequence>` tag, RGML allows `the <Iteration>` tag to embed:

- **The `<Exit>` tag**: Within this tag, the designer of the process captures the condition(s) that the requirements engineer must satisfy for the execution of the iteration to terminate.

4.1.4 Split Construct

The third RGML structural construct is the split construct, depicted in RGML by the `<Split>` tag. The designer uses this split construct to capture a group of activities from which the requirements engineer chooses only to execute. The activity the requirements engineer chooses resembles the concept of the "Dijkstra Guarded Commands," [Dijkstra 1975] because it can be based on the fulfillment of zero or more conditions. For terminology purposes, each option the requirements engineer has to execute is called a branch of the split. Figure 16 shows a diagram of a simple split structure, while highlighting the concept of a branch. The designer of the process can set conditions on one or more branches of the split. If the requirements engineer satisfies the condition(s) for more than one branch, then he or she has the choice of executing any of the branches that have satisfied conditions. If the requirements engineer does not satisfy the conditions for any of the branches of the process, then the execution stops and he or she is allowed to go back in the process until he or she satisfies one or more necessary condition.

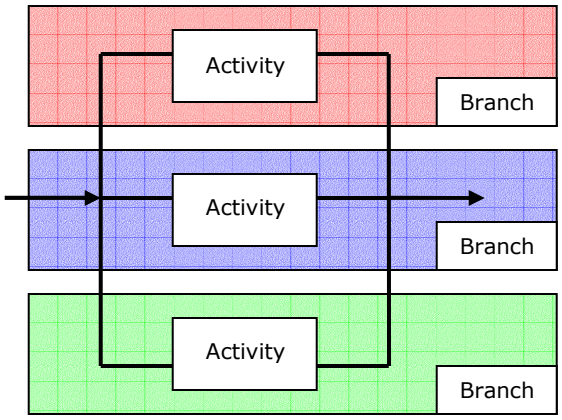

Figure 16. Split Structure with single activity branches

It is important for RGML to support the split structure, because in complex processes and in processes executed by humans it is common to see cases where there are a number of alternatives available. The ability to have conditions control the execution of branches increases the expressive power of RGML and gives RGML the ability to emulate other execution logic using a single construct, the split. For example, the designer can now express what is equivalent to an "if" condition by having a single branch with a condition. The designer can emulate the concept of a bypass for the path of execution by having all but one branch of a split contain conditions. As such if all the conditions of the split fail, the execution goes automatically through the bypass branch. RGML gives the designer flexibility and expressive power with the split construct, and it is left only to the designer's creativity to use its power to meet his or her needs.

In Figure 16, each branch of the split is composed of only one activity. However RGML supports multiple activities and/or structures on any branch. Figure 17, therefore shows a split structure in which the upper branch for that structure is an iteration structure, the middle branch is a sequence structure, and the last branch is a single activity.

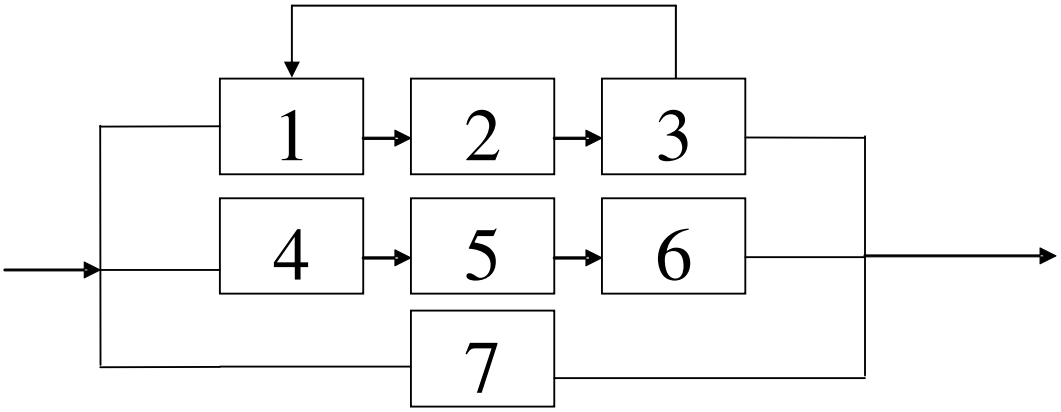

Figure 17. Split Structure with multiple activities on branches

Example 13, which shows how RGML expresses the structure seen in Figure 17, clarifies how RGML captures the conditions and structures of each branch in the split. RGML embeds all the information about a branch within the `<Branch>` tag.

```
<Split>
      <Branch name="Method A">
            <StartUseIf>
                  Conditions for the execution of this branch are here (See
                  Chapter 5 Section 2 for syntax of conditions)
            </ StartUseIf >
            <Sequence>
                  <RefActivity id=1>Number 1</RefActivity>
                  <RefActivity id=2>Number 2</RefActivity>
                  <RefActivity id=3>Number 3</RefActivity>
            </Sequence>
      </Branch>
      <Branch name="Method B">
            <StartUseIf>
                  Conditions for the execution of this branch are here (See
                  Chapter 5 Section 2 for syntax of conditions)
            </StartUseIf>
            <Sequence>
                  <RefActivity id=4>Number 4</RefActivity>
                  <RefActivity id=5>Number 5</RefActivity>
```

```
                <RefActivity id=6>Number 6</RefActivity>
            </Sequence>
        </Branch>
        <Branch name= "Method C">
            <Sequence>
                <RefActivity id=7>Number 7</RefActivity>
            </Sequence>
        </Branch>
</Split>
```

Example 13. Split Construct

As Example 13 shows, each `<Branch>` tag has a *`name`* attribute. Although it is unusual to give a branch in a split a name, RGML uses the name attribute of each branch when it needs to convey any information about the branch to the requirements engineer. The name attribute of the branch is the only way of identifying a branch to the requirements engineer. For example, RGML uses the *`name`* attribute of a branch to ask the requirements engineer which branch to execute if the conditions for more than one branch in the split are satisfied. Since it is not practical for RGML to communicate the branch to the requirements engineer as "the first branch," the need to have a name for each branch in the split is reasonable.

Each branch in the split consists of two sections. The first section is where the designer captures the conditions for the execution of the branch, which the `<StartUseIf>` tag denotes. Like the exit condition of an iteration construct, the conditions of the branches are expressed in a special format, which Chapter 5 Section 2 concentrates on describing. As in the third branch of the split in Example 13, if the `<Branch>` tag has no `<StartUseIf>` tag, it means that the designer does not want to associate any conditions with this branch and therefore, there are no restrictions for the execution of this specific branch.

The second section of the branch is the actual contents of the branch. If the branch consists of a group of activities in any specific structure, then the designer captures the

whole structure within the <Branch> tag. This format is seen in the iteration and sequence structures of Example 13.

RGML allows the designer to embed a number of tags within the <Sequence> and <Iteration> tags, but he or she can embed only the <Branch> tag within the <Split> tag. However, RGML allows the <Branch> tag to embed a number of tags. In addition to the <IncludeStructure>, <MilestoneScope> and <Group> tags explained with the <Sequence> tag, the <Branch> tag can also embed:

- **The <Sequence>, <Iteration>, <Split> or <Composite> tags:** The designer uses these tags to capture the structure of the activities within the branch. He or she can use one of these structures or a combination of them to express the activities of the branch.
- **The <StartUseIf> tag:** Within this tag, the designer of the process captures the condition(s) that the requirements engineer must satisfy before the process considers the branch as an option for execution. The conditions are based on questions, artifacts or milestones. The designer may join two or more conditions with Boolean operations to create a more complex condition. Chapter 5 Section 2 shows examples of conditions and how RGML is used to express them.

4.1.5. Composite

The last of RGML's basic constructs is the composite construct denoted by the <Composite> tag. The composite construct is different from the previous three constructs (the sequence, iteration and split), which the designer uses to express a specific arrangement of the activities. The designer uses the composite construct to capture an irregular arrangement of activities that he or she can not express using any combination of sequences, iterations or splits.

The importance of having the composite construct as one of the structural constructs of RGML comes from RGML's goal to have a language capable of expressing **any** requirements generation process. RGML needs a construct like the composite construct to provide the capability to characterize process structures that do not conform to sets of sequences, iterations and splits. To create a construct capable of expressing almost any arrangement of activities, RGML views and defines structures as a combination of start points, exit points and connections among sets of activities, where:

- **Start points** are activities through which it is possible to initiate the flow of control *into* a composite structure; and
- **Exit points** are activities through which the flow of control *exits* a composite structure; and
- **Connections** are used to define the flow of control between any activities *within* a composite structure.

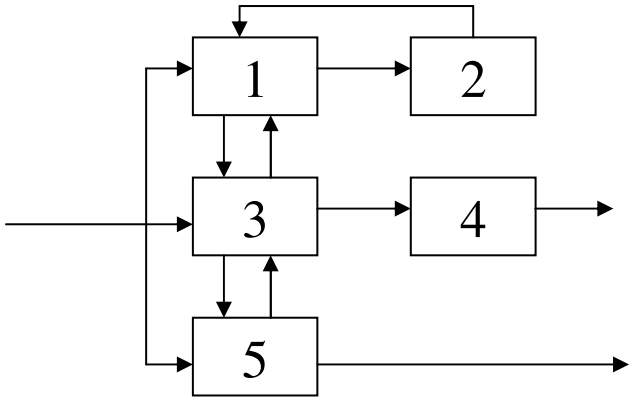

Figure 18. Irregular Flow Diagram

Like the branches of a split structure, RGML allows the designer to associate conditions with starting points. When the condition or conditions for a starting point are satisfied, then the requirements engineer can consider that as a point through which to enter the structure.

Defining a process structure such as the one Figure 18 depicts is difficult (if not impossible) using only the sequence, iteration and split constructs. While some components of the structure look "similar" to one the basic constructs (e.g., the iteration between activities 1 and 2) the presence of additional flows of control (e.g., from activities 1 and 3) complicate, or negate, the use of a basic construct. In Figure 18 it is possible to identify the starting points of the structure as activities 1, 3, and 5. The exit points from the structure are activities 4 and 5. The designer can easily characterize the connections between activities using sequence or iteration constructs. For example, activities 1 and 2, activities 1 and 3, and activities 3 and 5 are in iterations, while activities 3 and 4 are in a sequence. Example 14 shows how RGML expresses the structure depicted in Figure 18 using the composite construct.

```
<Composite>
   <StartPoints>
      <Entry>
         <StartUseIf> Conditions </StartUseIf>
         <RefActivity id=1> Number 1 </RefActivity>
      </Entry>
      <Entry>
         <StartUseIf> Conditions </StartUseIf>
         <RefActivity id=3> Number 3 </RefActivity>
      </Entry>
      <Entry>
         <StartUseIf> Conditions </StartUseIf>
         <RefActivity id=5> Number 5 </RefActivity>
      </Entry>
   </StartPoints>
   <Connections>
      <Iteration>
         <RefActivity id=1> Number 1 </RefActivity>
         <RefActivity id=2> Number 2 </RefActivity>
         :
      </Iteration>
```

```
      <Iteration>
         <RefActivity id=1> Number 1 </RefActivity>
         <RefActivity id=3> Number 3 </RefActivity>
         :
      </Iteration>
      <Iteration>
         <RefActivity id=3> Number 3 </RefActivity>
         <RefActivity id=5> Number 5 </RefActivity>
         :
      </Iteration>
      <Sequence>
         <RefActivity id=3> Number 3 </RefActivity>
         <RefActivity id=4> Number 4 </RefActivity>
      </Sequence>
   </Connections>
   <ExitPoints>
      <RefActivity id=4> Number 4 </RefActivity>
      <RefActivity id=5> Number 5 </RefActivity>
   </ExitPoints>
</Composite>
```

Example 14. Composite Construct

In Example 14, an `<Entry>` tag embedded within the `<StartPoints>` tag denotes each starting point. Within each `<Entry>` tag, there is a `<StartUseIf>` tag for the designer to use if he or she wants to associate any conditions with this particular starting point in the structure. The designer uses the `<RefActivity>` tag to identify the activity that he or she wants as a starting point. Further along in the code, is the `<Connections>` tag within which are embedded all the structural constructs that express a relationship between two or more activities. It is important to note that the order of the structural tags used under the `<Connections>` tag is not significant; the intent here is not to show the order of the structures, but to represent the different relationships between the activities within the structure. The last section of the code codifies the exit points of the structure. The designer achieves this by referencing the activities he or she considers as exit points under the `<ExitPoints>` tag.

4.1.6. Other Features of Structures

Sections 4.1.2 through Section 4.1.5 in this chapter concentrate on the syntax, semantics and importance of the sequence, iteration, split and composite constructs. Using these constructs, RGML is able to express the structure of the requirements generation process. However, while expressing the structure of such a process, there are other features the designer of the process might also need to capture. To increase the realism and to enhance the expressive ability of language, RGML provides three extra features the designer can use while expressing the structure of a process.

The first of the extra features allows the designer to group a number of structures together under one name. Although this feature does not add any functionality to the process, it is useful for the presentation of the process and for grouping activities into phases. For example, Figure 19 shows three activities, preparation, elicitation and evaluation which form a phase named Requirements Capturing. Figure 19 also shows another activity, Indoctrination, which when combined with the requirements capturing phase creates another phase named Requirements Gathering. Outside of these two phases there is an activity named Validation which is not part of any phase.

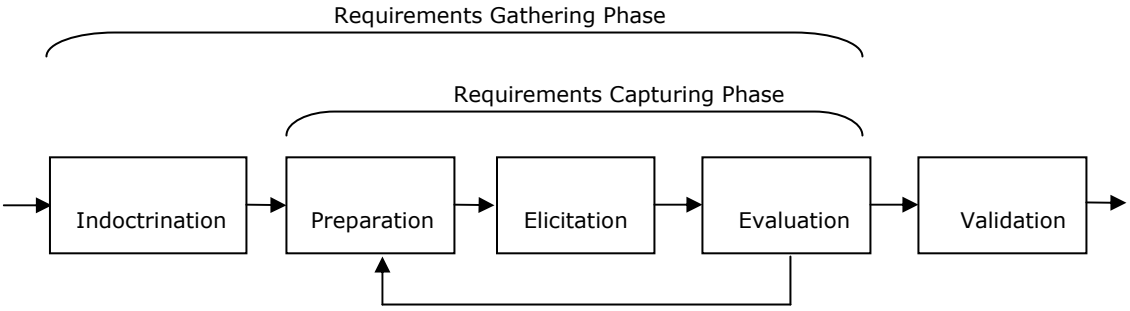

Figure 19. Different phases within a process

RGML allows the designer to embed all the structures he or she wants to group together between the opening and closing `<Group>` tags. Example 15 below shows how

RGML captures the process in Figure 19 by focusing on how the <Group> tag captures the different phases within the process.

```
<ProcessStructure>
      <Group name=”Requirements Generation”>
         <Sequence>
               <RefActivity id=”INDOC”> Indoctrination </RefActivity>
         </Sequence>
         <Group name= “Requirements Capturing”>
            <Iteration>
               <RefActivity id=”PREP”> Preparation </RefActivity>
               <RefActivity id=”ELICT”> Elicitation</RefActivity>
               <RefActivity id=”EVAL”> Evaluation</RefActivity>
            </Iteration>
         </Group>
      </Group>
      <Sequence>
            <RefActivity id=”VAL”> Validation</RefActivity>
      </Sequence>
</ProcessStructure>
```

Example 15. Grouping of Structures

Another of the extra features of RGML gives the designers the flexibility to make the execution of a certain structure or a group of structures optional. Despite the simplicity of this feature, it is important, because if the execution of all the structures in a process is obligatory, then this might hinder the requirements engineer's job more than facilitate it. When the designer of the process designates a structure as optional, and it is that structure's turn to execute, the system prompts the requirements engineer on whether he or she wants to continue with the execution of the structure or to skip its execution. This controlled flexibility that the designer of the process affords to the requirements engineer in executing the process is important for the success of the overall requirements generation process. When designers set too many guidelines and constraints on processes, the executers of the process tend to not follow the guidelines because of its rigidity. As a result, the whole process fails. The designer of the process needs to be careful with his or her use of this feature, because he or she might designate as optional a structure which

might be responsible for the production of an artifact the process requires for a later activity to continue. To declare a structure or group of structures or even an activity as optional the designer adds the "Optional" attribute and set its value to true. If the optional attribute is not present, then the system considers the structure, group of structures or activity as mandatory. Example 16 and Example 17 show how the designer adds the "Optional" attribute to structures and activities.

```
<RefActivity id="1" optional=true> Activity 1 </RefActivity>
```

Example 16. Adding an "optional" attribute to an activity

```
<Sequence optional=true>
      :
</Sequence>
```

Example 17. Adding an "optional" attribute to a structure

The third of the extra features of RGML provides the designer with the ability to pre-define reusable-structures and then include them in the code with only a reference. This feature assists the designer greatly when a certain structure is used more than once in a process, because it helps keep that reusable-structure in tact. At the same time, it decreases the clutter of code within the `<ProcessStructure>` section of RGML, thereby increasing the readability of the code. For example, if the designer of a process needs to use a specific sequence structure in several areas while defining the process-structure, instead of writing out the code of the sequence structure in every place it is needed, he or she defines the sequence structure in the Definitions section of the language. Example 18 shows how the designer can pre-define a reusable-structure:

```
<Definitions>
   <StructureLib>
      <DefineStructure id="F" name="Formatting Requirements">
         <Sequence>
            <RefActivity id=1> Activity 1 </RefActivity>
            <RefActivity id=2> Activity 2 </RefActivity>
            <RefActivity id=3> Activity 3 </RefActivity>
         </Sequence>
```

```
      </DefineStructure>
      :
   </StructureLib>
   :
</Definitions>
```

Example 18. Pre-defining a structure

Now instead of writing the code above in every location the designer uses the simple *IncludeStructure* command, shown in Example 19, to indicate that in this particular location the *IncludeStructure* command is to be replaced with the reusable-structure referenced within the command. Section 4.3 presents the concept of pre-defining reusable-structures and their reuse in more detail.

```
<IncludeStructure id="FReq">Formatting Requirements</IncludeStructure>
```

Example 19. IncludeStructure tag used to reference a pre-defined structure

In summary, this section presents the syntax and semantics of the constructs, such as sequence, iteration, split, and composite, which the designer of a requirements generation process uses with RGML to capture the structure of almost any requirements generation process. This section also discusses the three additional features of RGML, grouping of activities, making the execution of certain structures optional, and the concept of predefining structures for their reuse later in the language, which enhance its expressive power. The next section presents the syntax and semantics of the set of constructs the RGML uses to describe the specifics of the *activities* that are within a requirements generation process.

4.2 Activities

Section 1 in this chapter (4.1) presents the first component of RGML, process-structure, which is responsible for capturing the structure of the requirements generation

process. This section presents the syntax and semantics of the tags used within the second component of RGML, *Activities.* These tags capture the details of the activities the designer of the process references in the <ProcessStructure> component of the language. Example 20 shows the position of the *Activities* component relevant to the other components within RGML.

```
<RGML>
      <ProcessStructure>
            The flow of control of the activities is embedded here
      </ProcessStructure>
      <ActivityDetails>
            The description of the activities is embedded here
      </ActivityDetails >
      <Definitions>
            All the templates, artifacts, and types are embedded here
      </Definitions>
</RGML>
```

Example 20. Position of the Activities component relevant to the other RGML components.

4.2.1 Overview of the activities section

In the <ProcessStructure> section of the RGML, any existence of an activity is only a reference to the activity itself. No details about an activity are provided in the <ProcessStructure> section of the language. The design of the RGML forces the designer to define the details of all the activities that he or she references in the <ProcessStructure> section of the language in the activities section of the language. The RGML denotes the activities section of the language by the use of the <ActivityDetails> tag. The designer defines each activity in the process by the use of the <Activity> tag, which is embedded under the <ActivityDetails> tag, as shown in Example 21.

```
<ActivityDetails>
      :
      <Activity id="ID of Activity placed here">
            All the components RGML uses to capture the details of
            activities are embedded here
      </Activity>
      :
</ActivityDetails>
```

Example 21. Overview of the Activity's Section

Each activity has a unique *Id* throughout the process definition. This *Id* is the same one the designer uses when he or she wants to reference the activity in the process-structure section of the language. Activities are not simple entities; they are complex because of the numerous specifics and details which make up each activity. RGML, however, simplifies the process of capturing the details of activities by providing designation abstractions for five major components or perspectives, which support a complete description of almost any activity. These components are:

- **Name**: This component captures the name of the activity.
- **Goal**: This component captures the objective(s) of the activity.
- **Preconditions**: This component captures the conditions, if any, that need to be satisfied before the execution of the activity takes place.
- **Exceptions**: This component captures any exceptions to the default flow of steps in the activity. The process executes these exceptions if conditions the designer associates with each exception hold.
- **Steps**: This component captures the default set of steps for an activity that the requirements engineer needs to execute in order to reach the set goal of the activity.

RGML denotes each of the above components by a corresponding tag. All of these tags are embedded within the `<Activity>` tag in the order shown in Example 22.

```
<Activity id="">
      <Name>
      :
      </Name>
      <Goal>
      :
      </Goal>
      <Preconditions>
      :
      </Preconditions>
      <AllExceptions>
            <Exception>
            :
            </Exception>
      </AllExceptions>
      <DefaultFlow>
            <Step>
            :
            </Step>
            <Step>
            :
            </Step>
      </DefaultFlow>
</Activity>
```

Example 22. The layout of the code of an activity

Example 22 actually provides only the skeleton of the code that captures the details an activity. This example uses various tags for which the following sections of the chapter provide detailed explanations. Example 22 focuses on showing the *organization* of the different components of the activity in RGML.

Sections 4.2.2, 4.2.3 and 4.2.4 provide the details of the syntax, semantics and discusses the importance of the different components RGML uses to describe the details within an activity.

4.2.2 Name, Goal and Preconditions

Since there is not much to present about either the Name or the Goal components of the activity, this section presents both of them together. These two components describe the activity at a very high level. RGML denotes the Name component with the `<Name>` tag. The designer of the process writes the name of the activity between the opening and closing `<Name>` tags. Similarly the designer identifies the primary goal or goals of the activity between the opening and closing `<Goal>` tag of the activity. The goal of an activity is a high-level, text-based description of the objectives of the enclosing activity. The designer does not need any special formulas or methods to express the goal(s).

Preconditions are a set of conditions that the requirements engineer must satisfy before the process permits him or her to execute the activity. The designer can only specify the precondition in a special format under the `<Preconditions>` tag. As with the `<Exit>` tag of the iteration construct and the `<StartUseIf>` tag of the split and composite constructs, the designer expresses the conditions within the `<Preconditions>` tag as questions, artifact existence or milestones. The designer may join two or more conditions with Boolean operations to create a more complex condition. Chapter 5 Section 2 provides examples of conditions and how RGML is used to express them.

4.2.3. Activity Exceptions

After the designer of the process states the name, goal and preconditions of an activity, he or she needs to describe the set of steps within the activity that the requirements engineer needs to execute in order to fulfill the objective of the activity. However, before the designer defines the set of steps of the activity, he or she first identifies any exceptional cases where the requirements engineer might need to execute a set of steps which differs from the default set the activity dictates. Since each one of these exceptions is associated with a set of conditions, the requirements engineer does not follow the default set of steps if the condition or conditions the designer associates with the exception are satisfied.

For terminology purposes, *default scenario* refers to the default set of steps of the activity, while *alternative scenario* refers to a sets of steps associated with an exception. The designer captures the *default scenario* of the activity under the `<DefaultFlow>` tag, which Section 4.2.4 describes in more detail. The designer identifies each of the *alternative scenarios* of the activity within an `<Exception>` tag and embeds all the exceptions of an activity within the `<AllExceptions>` tag. Within each exception, the designer captures the condition that triggers the exception within the `<UseIf>` tag. Like all the other conditions in the language, RGML expresses these conditions in the special format as described in Chapter 5 Section 2 in more detail. Example 23, which presents the skeleton of the activity exceptions component, illustrates the different locations within the language for the *default scenario* and the *alternative scenarios*. The designer captures the *default scenario* under the `<DefaultFlow>` tag while he or she captures the *alternative scenarios* within the `<Flow>` tag of each activity exception. This is illustrated in Example 23.

```
<Activity>
   :
   <AllExceptions>
      :
      <Exception>
         <UseIf>
            Conditions for the use of this alternative scenario
         </UseIf>
         <Flow>
            An alternative scenario is here
         </Flow>
      </Exception>
      :
   </AllExceptions>
   <DefaultFlow>
      The default scenario of the activity is here
   </DefaultFlow>
</Activity>
```

Example 23. Framework of the exceptions component of an activity

RGML's support for the concept of activity exceptions adds a great deal of expressive power and flexibility to the language. Activity exceptions are helpful for capturing special cases of an activity. For example, the requirements engineer may want to execute an activity within an iteration in a certain way during the first execution of the iteration, but then needs to execute it in another way for the remaining iterations. Without RGML's support of the concept of activity exceptions, the only way to implement such a situation is to separate the activity into two different activities and change the structure of the process.

4.2.4 Steps of an Activity

The core of the description of any activity is the actual steps the executer of the activity has to go through to accomplish that activity. Section 4.2.3 introduces the notions of default scenario and alternative scenario and explains the difference between both. This section provides an explanation of how a designer captures the set of steps within an activity and the syntax that relates to those steps. Since the examples used in this section assume that the designer is capturing the steps of the *default scenario*, the code is embedded between the opening and closing `<DefaultFlow>` tags. The syntax to capture an *alternative scenario* is identical, except that the code is embedded between the opening and closing `<Flow>` tags of the exception.

RGML allows the designer to specify an unlimited number of steps for each activity. RGML identifies each step in the activity by a name, which the `<Name>` tag captures, and by a description, which the `<Description>` tag captures. The description of the step is a text based articulation of what the requirements engineer executing the step should do. Example 24 shows a sample of how the designer expresses a step within RGML.

```
<DefaultFlow>
      <Step Required=True>
            <Name> Meet with Stakeholders </Name>
            <Description> Call a meeting with all the stakeholders of
            the project and introduce yourself to them and become
            familiar with the role and job of each of the stakeholders
            and point out who are decision makers.</Description>
            :
      </Step>
</DefaultFlow>
```

Example 24. An activity step in RGML

RGML denotes each step in the activity by a `<Step>` tag. The `<Step>` tag has an attribute named "Required," which the designer sets to "true" if he or she wants to make sure that the requirements engineer executing the activity must execute this step. The option to have steps as required and others as not distinguishes between (a) the steps the designer thinks are crucial for the success of the activity and (b) those which the designer recommends the requirements engineer execute if he or she chooses to do so.

Since each step is different from other steps within the same activity and from other steps within different activities throughout the process, each step is unique. Some steps give directions to the requirements engineer on how to perform a certain action in real life, like conducting a meeting. For these kinds of steps a mere description of the step is sufficient, as Example 24 shows. For other steps which involve the requirements engineer in the production or use of artifacts, RGML launches word processing applications to display the artifacts or loads relevant document templates, if any, to help the requirements engineer create a new document. Another category of steps can suggest the execution of an external application to help the completion of this step. For these steps, RGML only instantiates the external application.

Although Example 24 shows an activity step as a description, steps often need to interact with the computational environment for such reasons as the production or display of an artifact or for the instantiation of an application. RGML captures any interaction with the computational environment through an `<Action> tag`. Within the `<Action>` tag the designer specifies the type of interaction along with a description of the interaction if he or she determines such a need exists. Example 25 shows illustrates the designer can express an action within a step using RGML

```
<Step Required=True>
      <Name> Meet with Stakeholders </Name>
      <Description> Call a meeting with all the stakeholders of the
      project and introduce yourself to them and become familiar with
      the role and job of each of the stakeholders and point out who
      are decision makers.</Description>
      <Action>
            <Type> E-mail </Type>
            <Description></Description>
      </Action>
</Step>
```

Example 25. An activity step including an action

Example 25 illustrates the use of the `<Action>` tag within a `<Step>` tag. The designer uses the `<Type>` tag to specify the *type* of interaction he or she wants this action to perform. The type of the interaction refers to the *application* the designer wants the process to instantiate when the requirements engineer executes the step. RGML leaves these types to the designer to define, thereby creating an unlimited number of possibilities for action types. The *definitions* section of RGML is where the action types are defined in detail. Section 4.3.4 provides the details of how the designer defines action types in the *definitions* section of RGML. The `<Action>` tag also contains a `<Description>` tag that the designer can use to include any notes or clarification about the action.

If an action produces or uses any documents, diagrams, reports or other artifacts, the designer should embed an additional `<Artifact>` tag within the `<Action>` tag. The designer uses the `<Artifact>` tag to associate an artifact with an action. An important issue arises if during the requirements generation process, the requirements engineer executes an activity more than once, and that activity has an action which deals with an artifact, e.g., activities within an iteration construct. For example, if an activity within an iteration construct has an action which creates a new document, it raises the issue of whether the process should create a new document each time the activity is executed or only the first time. To address this issue, the designer adds an attribute, *the mode of operation*, to the `<Action>` tag to indicate the relationship between the artifact and the action. The five modes of operation that the RGML supports are:

- **Create and Display (CD):** The designer uses this mode of operation when he or she wants the action within the activity to create the artifact the first time the requirements engineer executes the activity and display the artifact for any subsequent executions of the activity. With this mode, the RGML *does not* allow any modifications to the document once the requirement engineer creates it.

- **Create and Modify (CM):** The designer uses this mode of operation when he or she wants the action within the activity to create the artifact the first time the requirements engineer executes the activity and display the artifact for any subsequent executions of the activity. With this mode, the RGML *does* allow modifications to the document after the requirement engineer creates it.

- **Create Only (C):** The designer uses this mode of operation when he or she wants the action within the activity to create the artifact every time the requirements engineer executes the activity. To distinguish between the artifacts the process creates every time the requirements engineer executes the activity, the system appends the iteration number to the Id of the document produced.

- **Display Only (D):** The designer uses this mode of operation when he or she wants the action within the activity only to display the artifact every time the requirements engineer executes the activity. This mode of operation assumes that another previous action or activity produces the artifact before this action displays it. With this mode of operation the process deals with this artifact in a read-only manner. The system does not allow any modifications to the artifact.

- **Modify Only (M):** The designer uses this mode of operation when he or she wants the action within the activity to display the artifact and to allow modifications each time the requirements engineer executes the activity. This mode of operation assumes that another previous action or activity produces the artifact before this action displays it.

Example 26 illustrates the use of the *"Mode"* attribute, which identifies the relationship between the artifact and the action, as well as the use of the `<Artifact>` tag, which the designer employs to associate an artifact with an action. In RGML, each artifact has an *Id* which uniquely identifies the artifact throughout the process. When the designer associates an artifact with an action, he or she specifies the *Id* of the artifact as an attribute within the `<Artifact>` tag. Between the opening and closing `<Artifact>` tags is the artifact's textual name, which has no functionality of use to the process, except that it helps increase the readability of the code. Considering that the *mode of operation* for the action in the example is *Create Only*, every time the requirements engineer executes the activity, the system appends the iteration number the *Id* of the artifact, thereby producing the artifact *AG01* the first time the requirements engineer executes the activity and the artifact *AG02* the second time and so on.

```
<Flow>
      <Step Required=True>
            <Name> Meet with Stakeholders </Name>
            <Description> Call a meeting with all the stakeholders of
            the project and introduce yourself to them and become
```

```
            familiar with the role and job of each of the stakeholders
            and point out who are decision makers.</Description>
            <Action Mode=C>
                  <Type> E-mail </Type>
                  <Description></Description>
                  <Artifact id= "AG"> Agenda </Artifact>
            </Action>
      </Step>
<Flow>
```

Example 26. An activity step including an action and artifact

4.3 Definitions

In this chapter Sections 1 and 2 present the first two components of RGML, *Process-Structure* and *Activities*. While the designer captures the details of the activities in the *Activities* section of RGML, he or she makes reference to a number of artifacts, templates and action types, which he or she needs to define later in the *Definitions* section of the language. This section presents the syntax and semantics of the tags the designer uses to define the artifacts, templates and action types. Example 27 below shows the position of the *Definitions* component relevant to the other components within RGML.

```
<RGML>
      <ProcessStructure>
            The flow of control of the activities is embedded here
      </ProcessStructure>
      <ActivityDetails>
            The description of the activities is embedded here
      </ActivityDetails >
      <Definitions>
            All the templates, artifacts, and types are embedded here
      </Definitions>
</RGML>
```

Example 27. Position of the Definitions component relevant to the RGML.

4.3.1 Overview of the Definitions component

The first two components of RGML, *Process-Structure* and *Activities*, describe certain aspects of a requirements generation process. The *Definitions* component is the library or the repository of the language and does not capture any new aspects of the requirements generation process. It is, however, responsible for capturing the definitions of all the elements the designer uses in the *Process-Structure* and *Activities* components, as well as in parts of the *Definitions* component of the language. The elements the *Definitions* component defines are (a) the predefined structures the designer references in the *Process-Structure* component, (b) the artifacts the designer references in the *Activities* component, (c) the action types the designer references in the *Activities* component, (d) and the templates the designer references in the *Definitions* component. For example, in Example 26 the designer references two elements he or she needs to define in the *Definitions* component; these elements are: the *action type* **"E-mail"** and the *artifact* **"AG"** (Agenda).

RGML divides the *Definitions* component, denoted by the `<Definitions>` tag, into four components, each for defining a specific element:

- **Structure Library**: defines all the reusable structures the designer references in the `<ProcessStructure>` component. RGML denotes this component by the `<StructureLib>` tag.

- **Action Types**: defines all the action types the designer references while capturing the steps of activities in the `<ActivityDetails>` component. RGML denotes this component by the `<ActionType>` tag.

- **Artifacts**: defines all the artifacts (documents, diagrams, reports, and so forth) he or she references within the action associated with a step in an activity in the *Activities* component. RGML denotes this component by the `<Artifacts>` tag.

- **Templates**: defines all the templates he or she references while defining artifacts in the *Definitions* component. RGML denotes this component by the `<Templates>` tag.

Example 28 shows how the four main components of the *Definitions* section embedded within the `<Definitions>` tag.

```
<Definitions>
      <StructuresLib>
            :
      </StructuresLib>
      <ActionType>
            :
      </ActionType>
      <Artifacts>
            :
      </Artifacts>
      <Templates>
            :
      </Templates>
</Definitions>
```

Example 28. Overview of the Definitions component

4.3.2. Defining Reusable-Structures

The first component of the *Definitions* section is the *Structure Library*, in which the designer pre-defines all the structures he or she might need to reuse later in the process. RGML allows the designer to pre-define a structure and then reuse that structure later throughout the process definition, by using only the Id of pre-defined structure, instead of rewriting the entire code for the structure. RGML's support for defining

reusable-structures increases the maintainability of the structure's code and reduces the clutter of the code, thereby increasing readability.

The designer defines the reusable-structures using the `<DefineStructure>` tag. RGML embeds all the `<DefineStructure>` tags within the `<StructuresLib>` tag, with each `<DefineStructure>` having a name and a unique *`Id`*. The designer can use any of the basic structural constructs (*sequence*, *split*, *iteration* and *composite*) to define the reusable-structure within the `<DefineStructure>` tag just as he or she would define the structure of a group of activities under the `<ProcessStructure>` component of RGML. After defining the reusable-structure, the designer can reference the reusable-structure anywhere in the process by referencing its *Id* within an `<IncludeStructure>` tag. Example 18 is reproduced below to show how the designer can define a reusable-structure under the *Definitions* component of the language.

```
<Definitions>
   <StructureLib>
      <DefineStructure id=”F” name=”Formatting Requirements”>
         <Sequence>
            <RefActivity id=1> Activity 1 </RefActivity>
            <RefActivity id=2> Activity 2 </RefActivity>
            <RefActivity id=3> Activity 3 </RefActivity>
         </Sequence>
      </DefineStructure>
      :
   </StructureLib>
   :
</Definitions>
```

Example 18. Pre-defining a structure

Example 29 below shows how the designer can reference the reusable-structure defined in Example 18 at various positions while defining the overall structure of the process in the `<ProcessStructure>` component. The designer uses a simple reference to the reusable-structure instead of writing the code of the whole reusable-structure in each position the he or she desires to use it.

```
<ProcessStructure>
   <Split>
      :
   </Split>
   <IncludeStructure id="F">Formatting Requirements</IncludeStructure>
   <Sequence>
      :
   </Sequence>
   <Split>
      :
   </Split>
   <IncludeStructure id="F">Formatting Requirements </IncludeStructure>
</ProcessStructure>
```

Example 29. Referencing reusable-structures within the <ProcessStructure> Component

4.3.3. Defining action types

The second component within *definitions*, *Action Types*, is responsible for defining all the action types the designer uses when he or she is describing the details of the activities in the *Activities* component of RGML. Section 4.2.4 provides an explanation of the importance of RGML supporting the concept of *actions* and, subsequently, the concept of *action types*. RGML does not have a pre-defined set of *action types*; instead, in order to support the wide variety of action types, RGML leaves the designer to define all the action types. The designer does this by using the `<Type>` tag. Within the `<Type>` tag the designer specifies a unique Id for the *action type*. This is the same *Id* the designer uses to reference the *action type* in the activities component of the language. Between the opening and closing `<Type>` tags, the designer states the filename of the application he or she wants the system to instantiate when the requirements engineer executes an action of that type. There are cases, such as the "Data Flow Diagrams" in Example 30 where the designer does not associate any application with the *action type*. When the designer omits any association, the action type becomes only a

means to categorize actions and, in this case, the *action type* adds no functionality to the execution of the activity. Example 30 shows how RGML defines several *action types*.

```
<Definitions>
      :
      <ActionType>
         <Type id="E-mail"> C:\outlook.exe </Type>
         <Type id="Document"> C:\winword.exe </Type>
         <Type id="Spreadsheet"> C:\excel.exe </Type>
         <Type id="Presentation"> C:\powerpoint.exe </Type>
         <Type id="Data Flow Diagram"></Type>
      </ActionType>
      :
</Definitions>
```

Example 30. Defining different types of actions

4.3.4. Defining artifacts

The third component within *definitions*, *Artifacts*, is responsible for defining all the artifacts the designer uses when he or she is describing activity details in the *Activities* component of RGML. Since artifacts are important to any requirements generation process, there must be a library of all the artifacts the process uses in any of its phases. This library is the *Artifacts* component. The designer must define any artifact he or she uses anywhere in the process definition through the use of the `<Artifact>` tag. When defining an artifact, it is important for the `<Artifact>` tag to have the artifact's *Id* as an attribute, because this is the same *Id* that the designer uses whenever he or she references the artifact within the process definition. Within the `<Artifact>` tag are three other tags the designer uses to complete the definition of the artifact:

- **The `<Name>` tag:** The designer uses this tag to specify the name of the artifact he or she is defining.

- **The `<Template>` tag:** The designer uses this tag to associate a template with the artifact. The designer puts the *Id* of the template he or she wants to associate with

the artifact as the *Id* attribute of the `<Template>` tag. The designer can put a name for the template between the opening and closing `<Template>` tags, but this name has no significance. It only helps increase the readability of the code.

- **The `<Notes>` tag:** The designer can use this tag to add any notes or comment he or she finds important about the artifact, such as information about the use of the artifact, who should create it, what the artifact is used for, and so forth.

Example 31 shows how RGML is used to define one artifact within the *Definitions* component of RMGL.

```
<Definitions>
      :
      <Artifacts>
         <Artifact id=”ABCD”>
            <Name>Meeting Agenda</Name>
            <Template id=”Meeting”>Meeting Agenda Template</Template>
            <Notes></Notes>
         </Artifact>
      <Artifacts>
      :
</Definitions>
```

Example 31. Defining an artifact in RGML

4.3.5. Defining Templates

Templates, the fourth component within *Definitions*, is the component used when defining templates the designer references when he or she defines artifacts in the *Definitions* component of RGML. In general, templates are documents or files which have a preset format. RGML uses templates as a starting point for the creation of another document so that the requirements engineer creating the document does not have to recreate the format each time he or she creates the document. Since the designer of the

process is the one who creates these templates, he or she advises the requirements engineer to use these templates for better documentation of the requirements and to help with the standardization of the documents used throughout the process. The designer must use the `<Template>` tag to define any template he or she references when defining any artifact. The `<Template>` tag must have an *Id* as an attribute, which is the same *Id* the designer uses whenever he or she references this template when defining an artifact. Between the opening and closing `<Template>` tags the designer specifies the filename of the template document. Example 32 shows how RGML defines some *templates.*

```
<Definitions>
      :
      <Templates>
            <Template id=”Meeting”> c:\meeting.doc </Template>
            <Template id=”Report”> c:\report.doc </Template>
      </Templates>
</Definitions>
```

Example 32. Defining a template using RGML

4.4 Summary

In summary, RGML has three main components: *Process-Structure*, *Activities* and *Definitions*. The designer of the language uses the *Process-Structure* and the *Activities* components to capture the characteristics of a requirements generation process. The *Definitions* component of the language acts as a library or repository for all the pre-defined reusable-structures, actions, templates, and artifacts which the designer references throughout the process. The *Process-Structure* component of RGML uses four main constructs to capture the structure of any requirements generation process:

- **Sequence**: to capture a group of activities the requirements engineer executes in a sequential order

- **Iteration**: to capture a group of activities the requirements engineer executes in a loop
- **Split**: to capture a group of activities from which the requirements engineer chooses only one of the activities to execute
- **Composite**: to capture irregular structures by dividing them into entry points, connections between activities and exit points.

In addition to these four basic structure constructs, RGML supports grouping of activities, pre-defining reusable-structures, and the ability to declare any activity or structure in the process definition as optional. The *Activities* component of RGML captures the details of the activities of the process using five main sub-components:

- **Name** : captures the full name of the activity
- **Goal**: identifies the goal or goals of the activity
- **Preconditions** : captures the conditions which need to be satisfied before the execution of the activity takes place
- **Steps** : codifies the steps the activity recommends in order to reach the set goal of the activity
- **Exceptions**: captures an alternative set of steps (*alternative scenarios*) which the requirements engineer might execute instead of the original set of steps of the activity (*default scenario*), if certain conditions hold.

The *Definitions* component of RGML is responsible for capturing the definitions of four elements which are referenced throughout a process definition:

- **Reusable-Structures**: the reusable-structures that are referenced within the `<ProcessStructure>` component of RGML
- **Action Types**: the action types the designer references while capturing the details of the activities within the `<ActivityDetails>` component

- **Artifacts:** documents, diagrams, reports or any items the designer references while capturing the details of the activities within the `<ActivityDetails>` component
- **Templates**: documents which can be used as guidelines in the creation of artifacts and which the designer references when he or she is defining artifacts.

The features and elements of RGML in this chapter are its basic features, which describe the static side of the process and which are aspects of the process that do not change or deal with the execution of the process. RGML has more features which enhance its expressive capabilities and describe the dynamic side of the requirements generation process. Chapter 5 provides descriptions of these capabilities and the benefits of each, as well as an explanation of how each contributes to the expressiveness of RGML.

5. Supporting RGML's Expressive Capabilities

Chapter 4 provides a discussion of the elements and constructs that RGML uses to capture the structure and activities of a requirements generation process. RGML goes beyond only expressing the process-structure and the process's activities to express more complex concepts within a requirements generation process. This chapter provides descriptions of the concepts of *Conditions* and *Milestones* that enhance the expressive capabilities of RGML. Additionally, this chapter provides an explanation of how RGML expresses the *temporal aspect* of a process using *Conditions*, *Milestones*, and *Activity exceptions*.

5.1 Milestones

Milestones are elements which can assume only two values, "On" or "Off." The designer of a process can turn milestones "On" or "Off" at any location he or she wishes within the process. Milestones are generic indicators a designer can use to specify a number of different events. The reasoning behind using of a milestone depends on how and where the designer sets up the milestone. For example, the designer might set up a milestone to go "On" after the execution of an activity. This milestone, therefore, signals the completion of an activity. If the milestone is "On," then he or she has executed the activity; if the milestone's value id "Off," then he or she has not yet executed the activity. Figure 20 shows another situation which illustrates the use and importance of milestones.

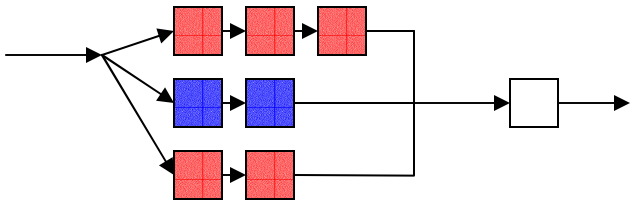

Figure 20. A sample process used to explain the concept of Milestones

In Figure 20, there is a split structure, which has three branches, and a single activity following the split structure. To explain how a milestone works, assume the

designer of the process in Figure 20 wants the single activity (located after the split) to execute an alternative scenario when the requirements engineer executes the middle branch (in Blue) of the split structure. Additionally the designer wants the single activity (located after the split) to execute the default scenario when requirements engineer executes either the first or the last branches (in Red) of the split. The designer can not capture the dynamics of the situation while *designing* the process because he or she does not know, at design time, which branch the requirements engineer will choose to execute. Therefore, the designer needs to set up a method by which the system can decide on its own at runtime (during the execution of the process) which activity scenario to execute based upon the execution path the requirements engineer decides to take. To achieve the desired dynamics the designer defines a milestone, M1, in the first activity of the middle branch of the split. The system switches milestone M1 *"On"* when the requirements engineer starts to execute the activities of the middle branch. To make the single activity execute the alternative scenario, the designer creates an *activity exception* with a condition that is only "true" if milestone M1 is *"On."* With this setup, when the requirements engineer executes the activities in the split's middle branch, the system triggers milestone M1 *"On,"* thereby making the condition for the activity exception "true," and the activity executes an alternative scenario. If the requirements engineer chooses any branch of the split other than the middle one, the milestone is "Off" by default and the condition for the activity exception fails, thereby leading the activity to execute the default scenario. Therefore, including the concept of milestones within RGML is necessary to give the designer the flexibility to capture the dynamics of the requirements generation process.

When the designer defines a milestone, he or she can state the *scope* of the milestone, the region within the defined process where the system maintains the value of the milestone. During the execution of the process, once the system exceeds the scope of a milestone, it immediately resets the value of the milestone to "Off," regardless of the current value of the milestone. A situation in which the designer makes use of the scope of a milestone occurs when he or she wants the system to automatically reinitialize the value for a milestone for each iterative pass through a sequence of activities. The

designer resolves the situation by setting the scope of the milestone within the iteration construct so that each time the requirements engineer finishes one iteration of the iteration structure, the system resets the milestone's value before the requirements engineer starts the execution of the new iteration. Example 33 illustrates how the designer defines the milestone scope for the sample process described above:

```
<ProcessStructure>
      <Iteration>
            <MilestoneScope id= "M1">
                  <RefActivity id= "1"> Activity 1 </RefActivity>
                  <RefActivity id= "2"> Activity 2 </RefActivity>
                  <RefActivity id= "3"> Activity 3 </RefActivity>
            </MilestoneScope id= "M1">
            <Exit>
            :
            </Exit>
      </Iteration>
</ProcessStructure>
```

Example 33. Defining the scope of a Milestone

When defining the scope of a milestone, the designer must give the milestone a unique *Id,* which he or she includes as an attribute of the `<MilestoneScope>` tag. For RGML to allow the scopes of different milestones to embed and intersect with each other, the closing tags of the `<MilestoneScope>` tag must include the *Id* of the Milestone in so that RGML understands which milestone the closing tag belongs to.

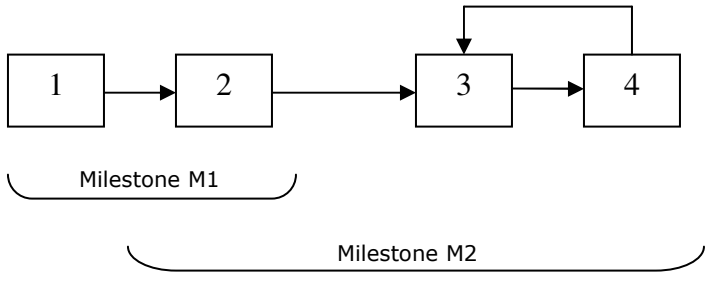

Figure 21. Sample Structure with overlapping milestones

Example 34 shows how designers can define overlapping milestones, as depicted in Figure 21, without any ambiguity.

```
<ProcessStructure>
      <Sequence>
            <MilestoneScope id= "M1">
                  <RefActivity id= "1"> Activity 1 </RefActivity>
            <MilestoneScope id= "M2">
                  <RefActivity id= "2"> Activity 2 </RefActivity>
            </MilestoneScope id= "M1">
      </Sequence>
      <Iteration>
                  <RefActivity id= "3"> Activity 3 </RefActivity>
                  <RefActivity id= "4"> Activity 4 </RefActivity>
            </MilestoneScope id= "M2">
            :
      </Iteration>
</ProcessStructure>
```

Example 34. Defining multiple intercepting Milestones

If the designer does not assign a scope to a milestone, then RGML automatically defines the scope of that milestone as the whole requirements generation process. As a result, during the entire execution of the process, the system will never reinitialize the value of the milestone.

After defining the scope of a milestone, the designer can trigger the milestones *"On"* or *"Off"* only within the `<Flow>` or the `<DefaultFlow>` tag of an activity. As Example 35 shows, the designer can trigger a milestone by using the `<Milestone>` tag. The designer must provide the *Id* of the milestone he or she wishes to trigger by including it as an attribute in the `<Milestone>` tag. The designer specifies the value he or she wants the milestone to have between the opening and closing `<Milestone>` tags.

```
<DefaultFlow>
      :
      <Milestone id= "M1">On</Milestone>
      <Step>
            :
      </Step>
      <Milestone id= "M2">Off</Milestone>
      <Step>
            :
      </Step>
      :
</DefaultFlow>
```

Example 35. Trigering Milestones *"On"* and *"Off"*

Example 36 shows how RGML uses milestones to capture the sample process illustrated in Figure 20, reproduced here with activity numbers to help with the explanation of Example 36.

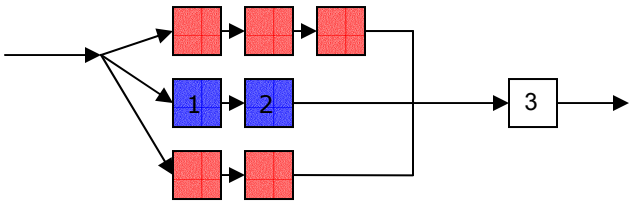

Figure 20 reproduced with activity numbers

In this particular situation the designer wants activity 3 to execute an alternative scenario, if the requirements engineer executes the middle branch of the split. Otherwise the designer wants activity 3 to execute the default scenario. To capture this situation, the designer must trigger a milestone "On" as soon as the requirements engineer executes the first activity in the middle branch (activity 1). Then the designer places a condition that executes an alternative scenario in activity 3 if the milestone is set "On." If the

milestone is set "Off," which means the requirements engineer did not execute the middle branch of the split, then activity 3 will execute its default scenario. Example 36 below shows how RGML permits a description of this situation:

```
<ProcessStructure>
      <Split>
            <Branch>
                  :
            </Branch>
            <Branch>
                  :
                  <RefActivity id= "1"> Activity 1 </RefActivity>
                  <RefActivity id= "2"> Activity 2 </RefActivity>
            </Branch>
            <Branch>
                  :
            </Branch>
      </Split>
      <Sequence>
            <RefActivity id= "3"> Activity 3 </RefActivity>
      </Sequence>
</ProcessStructure>
<ActivityDetails>
   <Activity id=1>
      :
      <DefaultFlow>
            :
            <Milestone id= "M1">On</Milestone>
            :
      </DefaultFlow>
      :
   </Activity>
   <Activity id=3>
      :
      <AllExceptions>
         <Exception>
            <UseIf>
```

```
                <Condition type="Milestone" Satisfied="On">M1</condition>
              </UseIf>
        </AllExceptions>
        :
    </Activity>
</ActivityDetails>
```

Example 36. Controlling the execution of an activity by milestones

Example 36 begins by showing how the designer defines the structure of this process under the <ProcessStructure> section of the language. In the <ActivityDetails> section, the designer sets a milestone "On" during the execution of activity 1 which is the first activity in the middle branch of the split. Then the designer creates an activity exception (an alternative scenario) within activity 3 and defines the condition for the execution of the activity exception. The condition is set so that it is "true" if milestone M1 is turned "On." Chapter 5 Section 2 gives a detailed explanation of conditions.

5.2 Conditions

In RGML, the designer must express any condition throughout a requirements generation process specification using a standard format, which this section concentrates on describing in detail; there are only five places defined in the language where the designer can use conditions.

5.2.1 Locations in RGML where conditions are used

The five locations available to the designer are:

1. Within the `<Precondition>` tag of an activity (see Example 37)

```
<Activity id="">
      :
      <Preconditions>
            Conditions are used here
      </Preconditions>
      :
</Activity>
```

Example 37. Use of conditions within the description of an activity

The requirements engineer must satisfy the conditions the designer uses within the `<Precondition>` tag before he or she can execute the activity. Chapter 4 Section 4.2.2 provides the detail of preconditions and their use within activities.

2. Within the `<Exit>` tag of an iteration construct (see Example 38)

```
<Iteration>
      :
      <Exit>
            Conditions are used here
      </Exit>
</Iteration>
```

Example 38. Use of conditions within the exit of an iteration

The requirements engineer must satisfy the conditions the designer uses within the `<Exit>` tag before he or she can exit an iteration construct. Chapter 4 Section 4.1.3 provides the details of exit conditions and their use within iterations.

3. Within the <UseIf> tag of an activity alternative (see Example 39)

```
<Activity>
      :
      <Exceptions>
            <Exception>
                  <UseIf>
                        Conditions are used here
                  </UseIf>
                  :
            </Exception>
            :
      </Exceptions>
</Activity>
```

Example 39. Use of conditions within the exception tag of an activity alternative

The system must satisfy all the conditions the designer uses within the <UseIf> tag before it executes the activity exception. Chapter 4 Section 4.2.3 provides the details of activity exceptions.

4. Within the <StartUseIf> tag of a branch in the split construct (see Example 40)

```
<Split>
      <Branch>
            <StartUseIf>
                  Conditions are used here
            </ StartUseIf >
            :
      </Branch>
      :
</Split>
```

Example 40. Use of conditions within the branch of a split construct

The requirements engineer must satisfy the conditions the designer uses within the <StartUseIf> tag before he or she can execute that branch of the split. Chapter 4 Section 4.1.4 gives the details of split constructs.

5. Within the <StartUseIf> tag of an entry of the composite construct (see Example 41)

```
<Composite>
      <StartPoints>
            <Entry>
                  <StartUseIf>
                  Conditions are used here
                  </StartUseIf>
                  :
            </Entry>
            :
      </StartPoints>
      :
</Composite>
```

Example 41. Use of conditions within the entry of a composite structre

The requirements engineer must satisfy the conditions the designer uses within the <StartUseIf> tag before the system allows him or her to start execution (entry point) of the structure through this activity. Chapter 4 Section 4.1.5 provides the details of composite constructs.

These five positions are the only places within the language where RGML allows the designer to use conditions. In any of these positions, the designer captures the conditions using the same format. The following simple scenario provides the best explanation of the format RGML uses to express condition.

5.2.2 The syntax of conditions

This simple scenario consists of the following condition: within an iteration construct, the designer of the process wants to ensure that the requirements engineer meets with all the stakeholders before he or she can exit from the iteration. The designer captures this scenario in RGML as follows:

```
<Iteration>
      :
      <Exit>
            <Condition type="Question" Satisfied="Yes"> Have you met
            with all the stakeholders of this project </Condition>
      </Exit>
</Iteration>
```

Example 42. A simple condition expressed within the exit of an iteration

The code in Example 42 implies that if the requirements engineer answers "Yes" to the "Question" between the opening and closing condition tags ("Have you met with all the stakeholders of this project?"), then the condition is satisfied. As Example 42 shows, the `<Condition>` tag consists of three main parts:

- The "Type" attribute
- The "Satisfied" attribute
- The Body of the Condition (what is between the opening and closing condition tags)

The first part of the condition tag is the "Type" attribute, whose value depends on which entity the condition is based on. This attribute can assume one of the following values:

- **Artifact** : if the condition is based on the status of an artifact within the process
- **Milestone**: if the condition is based on the status of a milestone within the process
- **Question**: if the condition is based on the answer to a question that the system presents to the requirements engineer.

The second part of the `<Condition>` tag is the "Satisfied" attribute, which shows the value that the entity the condition is based on needs to hold for the condition to be "true". The set of values the "Satisfied" attribute can hold depends on the value of the

"type" attribute of the condition. Each value for "type" offers a set of values for the "Satisfied" attribute:

If the value of the "type" attribute is "**Artifact**" then the values for the "Satisfied" attribute can be:

- **Produced**: This value implies that the condition is "true" if the process creates the artifact the designer references in the condition.
- **Not Produced**: This value implies that the condition is "true" if the process has not yet created the artifact the designer references in the condition.

If the value of the "type" attribute is "**Milestone**" then the values for the "Satisfied" attribute can be:

- **On**: This value implies that the condition is "true" if by this point in the process the milestone the condition specifies is set *"On."*
- **Off**: This value implies that the condition is "true" if by this point in the process the milestone the condition specifies is set *"Off."*

If the value of the "type" attribute is "**Question**" then the values for the "Satisfied" attribute can be:

- **Yes**: This value implies that the condition is "true" if the requirements engineer answers *"Yes"* to the question the system presents to him or her.
- **No**: This value implies that the condition is "true" if the requirements engineer answers *"No"* to the question the system presents to him or her.

Since RGML supports only questions that the requirements engineer can answer with *"Yes"* or *"No,"* no open ended questions are permissible.

The third part of the condition tag, the body, differs depending on the value of the "type" attribute of the condition. If the condition is based on an artifact, then the body of the condition contains the *Id* of the artifact. Likewise, if the condition is based on a milestone then the body of the condition contains the *Id* of the Milestone. If the condition

is based on a question, then the body of the condition tag contains the text of the question the system uses to prompt the requirements engineer.

5.2.3 Constructing compound conditions

The syntax above shows how to express a single condition. However in many cases there is a need to express a combination of single conditions together in the form of one complex condition. Single conditions are joined together by means of the *"and"* or *"or"* Boolean operators. To combine two conditions, the designer embeds both conditions under the tag of the joining Boolean operator. Example 43 shows how RGML combines two conditions with an *"and"* operator.

```
<AND>
      <Condition ….. >
      <Condition ….. >
</AND>
```

Example 43. Combining two conditions

By replacing the `<AND>` operator with an `<OR>` operator, the condition becomes joined by an *"or"*. The simple manner in which RGML implements the combination of two or more conditions brings great power and flexibility to express even more complex combinations of conditions. The following set of examples give a better understanding of how RGML combines conditions.

Compound Condition:
(Condition 1 **and** Condition 2) **or** Condition 3

RGML representation of the Compound Condition:

```
<OR>
      <AND>
            Condition 1
            Condition 2
      </AND>
      Condition 3
</OR>
```

Example 44. Combining multiple conditions

RGML does not base precedence on the actual Boolean operators; instead it bases precedence on the embedding of the operators within each other. RGML embeds the operator of the highest precedence in the inner most level. In Example 44, since the "and" operation has higher precedence than the "or" operation, the designer embeds the two conditions combined by an "and" operator within the "or" operator.

Compound Condition:
Condition 1 **and** (Condition 2 **or** Condition 3) **and** Condition 4

RGML representation of the Compound Condition:

```
<AND>
      Condition 1
      <OR>
            Condition 2
            Condition 3
      </OR>
      Condition 4
</AND>
```

Example 45. Combining multiple conditions

In Example 45, the designer wants to combine *condition 2* and *condition 3* with an "or" operator and then combine the result of the previous combination with two other conditions: *condition 1* and *condition 4*. RGML can support the combination of more than two conditions under a single operator, as this example demonstrates. The result of the combination of *condition 2* and *condition* 3 along with *condition* 1 and *condition* 4 are all combined under an "and" operator.

Compound Condition:
(Condition 1 **and** (Condition 2 **or** Condition 3)) **or** Condition 4

RGML representation of the Compound Condition:

```
<OR>
   <AND>
         Condition 1
         <OR>
               Condition 2
               Condition 3
         </OR>
   </AND>
   Condition 4
</OR>
```

Example 46. Combining multiple conditions

Example 46 illustrates how RGML excels in handling precedence between several operators. In accordance to RGML's rule that embeds the operator of the highest precedence at the inner most level, the "or" operator between *condition 2* and *condition* 3 receives the highest precedence and is executed first. The designer then combines the result of the "or" operation with *condition 1* by an "and" operator. The he or she combines the result of that "and" operation with *condition 4* with another *"or"* operator.

Chapter 5 Section 5.2.2 shows how to use RGML to express a single condition. It also shows the different parts associated with the <Condition> tag. Chapter 5 Section 5.2.3 shows how RGML uses Boolean operators to combine two or more conditions. What follows is an attempt to combine the information in both these sections to show, with a set of examples how RGML captures multiple conditions and joins them together. These examples represent an effort to show the flexibility and power RGML offers in expressing conditions. Each of the examples below starts with a condition expressed in textual format and then shows how RGML captures that condition.

Condition expressed in textual format:

Terminate the iteration if the requirements engineer has interviewed all the stakeholders, **or** if the document ABCD has already been produced.

Condition expressed in RGML:

```
<Iteration>
   :
   <Exit>
      <OR>

      <Condition type="Question" Satisfied="Yes"> Have you
      interviewed all the stakeholders? </Condition>

      <Condition type="Artifact" Satisfied="Produced> ABCD
      </Condition>

      </OR>
   </Exit>
</Iteration>
```

Example 47. Multiple Conditions expressed in RGML

Example 47 illustrates a typical situation in which the requirements engineer can not exit from an iteration structure unless he or she fulfills a set of conditions. In this situation, to terminate the iteration the requirements engineer must answer *"yes"* to the question "Have you interviewed all the stakeholders?," or he or she must have created the document, *"ABCD"*, during some previous activity in the process. The designer captures the situation by joining both conditions within an <OR> tag and then embeds the entire

condition block (everything between the opening and closing <OR> tags) within the <Exit> tag of the iteration structure.

Condition expressed in textual format:

Start the activity if the document XYZ has not been produced yet and if the activity A has already been executed (depicted by Milestone A) or start the activity if the requirements engineer has not finished gathering all the requirements.

Condition expressed in RGML:

```
<Activity>
      :
      <Precondition>
         <OR>
            <AND>
               <Condition type="Artifact" Satisfied="Not
               Produced"> XYZ </Condition>

               <Condition type="Milestone" Satisfied="ON"> A
               </Condition>
            </AND>

            <Condition type="Question" Satisfied="No"> Have you
            gathered all the requirements </Condition>

         </OR>
      </Precondition>
      :
</Activity>
```

Example 48. Multiple Conditions expressed in RGML

Example 48 illustrates the precondition of an activity. The system does not allow the requirements engineer to start the execution of an activity unless he or she satisfies the activity's preconditions. The requirements engineer is allowed to execute the activity only if *Milestone A*, which indicates the completion of *Activity A*, is set "On" and if RGML makes sure that a document named XYZ has not been produced previously during the process. The system checks these two conditions before checking any other conditions, because these conditions are embedded within the inner most Boolean operator, which is the "And" operator. If the result between the two joined conditions is "true" then the system allows the requirements engineer to start the execution of the

activities without even considering any of the other conditions, because since the final result of the whole condition is already "true". However, if the result of the joining of both the milestone and the artifact conditions is "false" then the system asks the requirements engineer the question "Have you gathered all the requirements." Only if he or she answers "No" to the question does the system allow the requirements engineer to execute the activity.

Condition expressed in textual format:

Start the use of this branch of the split if *Activity* B has not been executed yet (depicted by Milestone B) or if the requirements engineer has not documented all the requirements. However in either case, the requirements engineer can not use this branch unless document ABC and document MNO have already been produced.

Condition expressed in RGML:

```
<Split>
   <Branch>
      <StartUseIf>
         <AND>
            <OR>
               <Condition type= "Milestone" Satisfied="Off"> B
               </Condition>

               <Condition type= "Question" Satisfied="No"> Have
               you documented all the requirements yet?
               </Condition>

            </OR>

            <Condition type="Artifact" Satisfied="Produced">
            ABC </Condition>

            <Condition type="Artifact" Satisfied="Produced">
            MNO </Condition>

         </AND>
      </ StartUseIf >
      :
   </Branch>
   :
</Split>
```

Example 49. Multiple Conditions expressed in RGML

Example 49 shows the conditions that guard the execution of a certain branch within a split structure. For the system to consider the execution of this branch, it must make sure that either *Milestone B* is "Off" or that the requirements engineer answers *"No"* to the question "Have you documented all the requirements yet?" If none of these conditions are satisfied, then the system does not consider any of the other conditions within the condition block and assigns the resultant of the whole condition block to "false." The system does so because the outer most condition is an *"and"* and all the conditions within an *"and"* must hold "true" for the whole result to hold "true." If one condition fails, the system does not consider any of the other conditions. This technique of execution saves time and computational power. If one or both of the previous conditions (Question or Milestone) are "true" then the system checks that the documents ABC and MNO are already created before the it assigns a "true" value as a resultant of the whole condition block.

5.2.4 "Hard and Soft" Conditions

There remains one last concept associated with conditions in RGML, *"Hard and Soft"* conditions. If the designer designates a condition as *"*Hard*,"* then the requirements engineer can not bypass the condition if he or she does not satisfy it and must go back in the process and satisfy whatever conditions were not fulfilled in order to continue the process. On the other hand, if the designer designates a condition as *"*Soft*,"* then, if the condition fails, the requirements engineer has the option of either bypassing the condition and continuing the execution of the process or going back in the process and satisfying whatever conditions are needed. A designer uses *"*Soft*"* conditions if he or she wants the requirements engineer executing the process to be observant of certain conditions as precautions, not as restrictions on the execution of the process. The designer of the process decides which conditions should be *"*Soft*"* and which should be *"*Hard.*"* Since all conditions in RGML are by default *"Hard"* conditions, there is no syntax needed to specify that a condition is *"*Hard.*"* However, if the designer wishes to specify a *"*Soft*"* condition, he or she adds the *"*Soft*"* attribute within the opening tag of the condition. Example 50 shows such a condition:

```
<Condition type="Artifact" Satisfied="Produced" Soft> ABC </Condition>
```

Example 50. "Soft" attribute for a condition

Although RGML's ability to permit "Soft" conditions provides additional flexibility to the requirements engineer executing the process, the designers must still use caution in order not to sacrifice the quality of the process when deciding which conditions are "Soft" and which are "Hard".

RGML captures conditions using a very powerful and flexible mechanism which gives the designer the ability to capture almost any condition a process might need. Section 5.3 provides an explanation of how a designer can use conditions, milestones and activity exceptions to represent temporal aspects of a requirements generation process.

5.3 Capturing the temporal aspect of a process

RGML has the ability to capture the structure of requirements generation processes using its set of basic constructs (sequence, iteration, split, and composite), described in detail in Chapter 4, Section 4.2. However, for RGML to express certain situations within the temporal framework of a process it needs the help of other concepts and elements, such as conditions and milestones, explained in Sections 5.1 and 5.2 of this Chapter. This section concentrates on defining the temporal aspect of a process, as well as explaining how RGML combines the use of many concepts and elements in order to capture the temporal aspect of a process.

5.3.1 Defining the temporal aspect

The temporal aspect of a process is the execution of the process with respect to time, while the structural aspect of a process shows the arrangement of the activities of a process. Figure 22 and Figure 23 show the difference between the temporal and structural aspects of a group of activities the requirements engineer executes three times. For the structural aspect, the designer puts the activities in an iteration construct (Figure 22). For

the temporal aspect, the designer lays out the activities with respect to time, as the requirements engineer **executes** them (Figure 23). The temporal aspect of a process is, therefore, the **unfolding** of the structure of the process (in Figure 22) over a period of time, which is the path the requirements engineer takes while executing the process. The structural aspect of a process involves the use of **activities**, while the temporal aspect involves the use of **activity occurrences**. Each *activity occurrence* is an instance of its corresponding activity during execution. There is a possibility that there might be differences between activity occurrences even though they originate from the same activity. For example, the designer of the process might want the requirements engineer to execute Activity 2 the second time with some minor differences from the other times he or she executes it. Therefore, it is important for a language, such as RGML, which aims at capturing a requirements generation process, to have some mechanism by which it can capture the activity occurrences of a process, thereby capturing specific scenarios within the temporal aspect of an activity

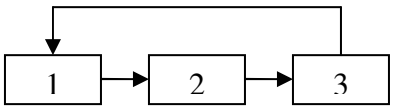

Figure 22. The Strcutral aspect of an iteration

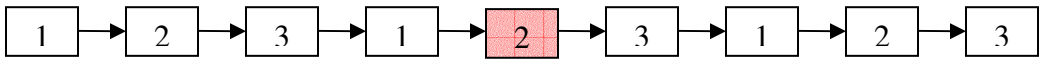

Figure 23. The Temporal aspect of an iteration

5.3.2 How RGML captures the temporal aspect of a process

RGML captures the structural aspect of a process by using structural constructs such as sequences, iterations, splits and composites. Since the temporal aspect of the process is inherent to the execution of the structure of the process, neither RGML, nor any other language, is capable of capturing the temporal aspect of a process, because it is unknown at the design time of the process. However, RGML is capable of capturing certain pre-defined scenarios within the temporal aspect of a process. To achieve this

challenging objective RGML must have a mechanism to describe certain activity occurrences within the process. RGML achieves this objective by using three concepts: *milestones*, *conditions* and *activity exceptions*.

The designer first sets up a milestone or a set of milestones that go "On" at a position before the activity occurrence. It is necessary to recall this previous explanation of how the designer wants the requirements engineer to execute activity 2 the second time with some minor differences from the other times he or she executes the activity in the process (Figure 23) in order to describe how the designer sets up milestones.

The positioning of milestones is crucial in order to capture an activity occurrence correctly. If the designer triggers the milestone "On" after the execution of activity 1 then when the system executes activity 2 the first time, it would find the milestone "On" and thereby executing the alterative scenario the first time activity 2 is executed, but not the second time. However, the correct position of the milestone is either after activity 2 or after activity 3, because in both cases the milestone is "Off" before the first time the requirements engineer executes activity 2 and "On" before the second time he or she executes the activity (See Figure 24).

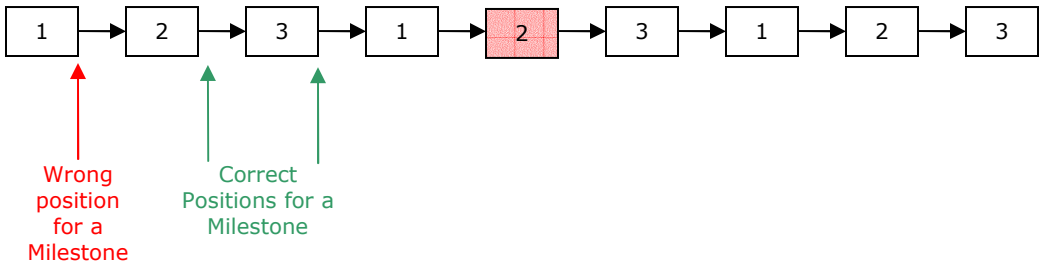

Figure 24. Possible positions for a Milestone

Assuming that in the previous explanation the designer chooses the position of the milestone to be after the execution of activity 2, Example 51 shows how the designer sets up the milestone while he or she is capturing the steps of the default scenario of that activity:

```
<ActivityDetails>
      <Activity id=2>
            :
            <DefaultFlow>
                  :
                  <Milestone id=M1>On</Milestone>
            </DefaultFlow>
      </Activity>
      :
</ActivityDetails>
```

Example 51. Setting up a Milestone

After the designer sets up the Milestone, if the Milestone is *"Off"* then he or she knows that the requirements engineer is executing activity 2 for the first time, but if the Milestone is *"On,"* then he or she knows that it is the second time the requirements engineering is executing activity 2. Therefore, the next step is for the designer to capture what he or she wants the requirements engineer to do when executing the activity 2 for the second time in an alternative scenario by creating an *activity exception* for activity 2. The condition for the designer to execute the activity exception is to find the status of the Milestone M1 *"On."* Example 52 shows how the designer sets up the activity exception:

```
<Activity id=2>
   :
   <AllExceptions>
      <Exception>
         <UseIf>
            <Condition type="Milestone" Satisfied="On">M1</Condition>
         </UseIf>
         <Flow>
            The alternative scenario is here
         </Flow>
      </Exception>
```

```
   </AllExceptions>
   <DefaultFlow>
      The default scenario of the activity is here
   </DefaultFlow>
</Activity>
```

Example 52. Setting up an activity excpetion

By having the designer setup the *activity exception*, he or she now successfully captures a **special set of steps** for the requirements engineer to execute on the **second occurrence** of the execution of the activity. This example illustrates how RGML is capable of capturing certain scenarios within the temporal aspect of a process by the use of *milestones*, *conditions* and *activity exceptions*. The next section provides a complex example to illustrate the ability of RGML to capture the temporal aspect of processes that are difficult to capture with a normal process description language.

5.3.3 An extended example

Figure 25 shows a sample process and Figure 26 shows the path the requirements engineer takes during the execution of the process. However the designer of the process has to accommodate three constraints to the execution of the process:

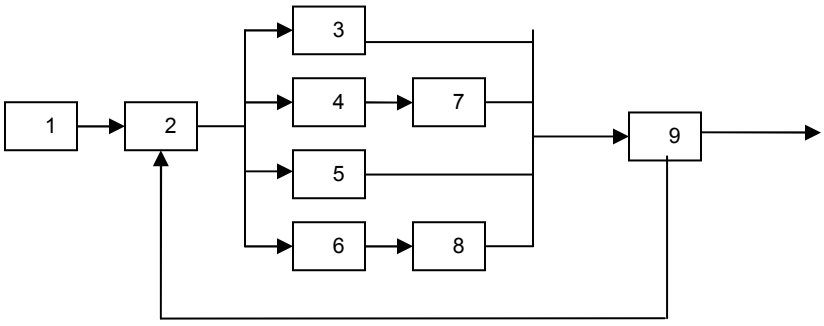

Figure 25. The structure of a Sample Process

- **Constraint 1:** to execute an alternative scenario for activity 2 **after activity 3** is executed for the **first time**.
- **Constraint 2:** to execute an alternative scenario for activity 3 the **second time** the activity is executed.

- **Constraint 3:** to execute an alternative scenario for activity 9 the **first time** the activity is executed.

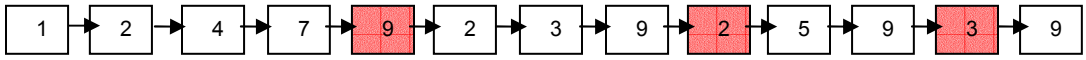

Figure 26. A possible execution path (temporal aspect) of the previous sample process

First, the designer analyzes the problem to decide where he or she is going to set up the milestones and then to solve the problem sets up two milestones as follows:

- **Milestone M3:** This milestone is set *"On"* after the first time the requirements engineer executes activity 3. This Milestone is also set *"Off"* after the second time the requirements engineer executes activity 3. This milestone is set up to help solve constraints 1 and 2.
- **Milestone M9:** This Milestone is set *"On"* after the first time the requirements engineer executes activity 9. This milestone is set up to help solve constraint 3

After the designer sets up the milestones, he or she now must create activity exceptions and set up the conditions for their execution. To meet the condition of constraint 3, the designer creates an activity exception for activity 9 that the system executes only if milestone M9 is *"Off."* Hence, when the requirements engineer executes activity 9 for the first time and sets milestone M9 *"On,"* then for the remaining times he or she executes the activity the value for milestone M9 would be *"On,"* thereby causing the failure of the condition to execute the activity exception and the execution of the default scenario of the activity. Since milestone M3 is set *"On"* after the first time activity 3 is executed, to meet the condition of constraint 1 the designer creates an activity exception for activity 2 that the system executes only if milestone M3 is *"On."* Since milestone M3 is set *"On"* after the first execution of activity 3, the designer can also use this milestone to meet the condition of constraint 2. To do this the designer creates an activity exception for activity 3 that the system executes only if milestone M3 is *"On."* In order to ensure that the system executes the default scenario for the activity

after the second time the activity is executed, the designer sets milestone M3 "Off" inside of the activity's alternative scenario. Example 53 shows how the designer expresses all the milestones, activity exceptions, and conditions in RGML.

```
<Activity id=9>
      :
   <AllExceptions>
      <Exception>
         <UseIf>
            <Condition type="Milestone" Satisfied="Off">M9</Condition>
         </UseIf>
         <Flow>
            The alternative scenario is here
            :
            <Milestone id=M9>On</Milestone>
         </Flow>
      </Exception>
   </AllExceptions>
   <DefaultFlow>
      The default scenario of the activity is here
   </DefaultFlow>
</Activity>
<Activity id=2>
   :
   <AllExceptions>
      <Exception>
         <UseIf>
            <Condition type="Milestone" Satisfied="On">M3</Condition>
         </UseIf>
         <Flow>
            The alternative scenario is here
         </Flow>
      </Exception>
   </AllExceptions>
   <DefaultFlow>
      The default scenario of the activity is here
   </DefaultFlow>
```

```
</Activity>
<Activity id=3>
   :
   <AllExceptions>
      <Exception>
         <UseIf>
            <Condition type="Milestone" Satisfied="On">M3</Condition>
         </UseIf>
         <Flow>
            The alternative scenario is here
            :
            <Milestone id=M3>Off</Milestone>
         </Flow>
      </Exception>
   </AllExceptions>
   <DefaultFlow>
      The default scenario of the activity is here
      :
      <Milestone id=M3>On</Milestone>
   </DefaultFlow>
</Activity>
```

Example 53 . RGML code using Milestones to satisify constraints on a process

This example illustrates the power of the expressive capabilities of RGML. RGML is able to capture three constraints for a process by the addition of only five additional lines of code manipulating milestones and capturing conditions. The concepts of activity exceptions, milestones and conditions help RGML express the temporal aspect of requirements generation processes, thereby, making RGML a complete language capable of capturing the static and dynamic aspects of almost any requirements generation process.

5.4 Summary

In summary RGML uses milestones and conditions to enhance its expressive capabilities. Milestones are generic indicators, the designer can trigger "On" or "Off" to indicate a number of different events. The designer can define a scope for a milestone within the process-structure component of RGML. Within the `<Flow>` or `<DefaultFlow>` tags of an activity, the designer can trigger milestones "On" or "Off." The primary use for milestones is within conditions. Conditions in RGML are captured using a flexibility and powerful format. Conditions can be based on questions that are prompted to the requirements engineer, or milestones, or artifacts, or any combination of these three elements. Compound conditions in RGML are joined together using the "and" or the "or" Boolean operator. The designer can also set conditions to be either "Soft" or "Hard" conditions. A unique capability in RGML is its ability to capture certain scenarios within the temporal aspect of a process, using milestones, conditions and activity exceptions. This chapter highlights the importance of the rich set of concepts that enhance the expressive capabilities of RGML. These enhancements enable RGML to capture dynamic aspects of requirements generation processes in addition to distinguishing RGML from other process description language.

6. Summary and Future Work

This thesis describes a solution to a problem in the field of requirements engineering with its introduction and explanation of RGML. Chapter 1 provides an introduction to the problem and to the Requirements Generation Markup Language (RGML). Chapter 2 concentrates on the relevant research in the areas of specification languages, process description languages, and markup languages which leads to the conclusion that RGML is the first specialized XML based process description language designed to formally specify the requirements generation process. Chapter 3 gives an overview of RGML and highlights the three main components of the language: process-structure, activities and definition. Chapters 4 and 5 show the different constructs RGML uses to describe the structure of the process, the activities within the process, and how to define conditions, milestones and activity alternatives. This chapter provides a summary of this work and presentation of the future work anticipated for this project.

6.1 Summary

The absence of any process description language which can formally specify and characterize a requirements generation process has motivated the work on RGML. This language is needed as a first step in creating an interactive environment which guides the requirements engineer through a requirements generation process. The creation of such an environment can help solve several problems that are currently visible in the field of requirements engineering, including volatile requirements, poor requirements, unpredictable requirements, and others. However, there are many challenges in developing a language that has as its objective the capability of specifying and describing requirements generation processes in a way that guides and assists the requirements engineer through the process. Some of these challenges are:

- How to describe the structure of the process
- How to express the necessary steps in an activity
- How to express the flow of execution of the process

- How to describe when and where to instantiate tools and applications in the process
- How to express the conditions within the process
- How to describe the wide variety of requirements generation processes using one mechanism.

To overcome these challenges, RGML has to be designed to include a rich set of constructs and concepts that give the language expressiveness and flexibility. RGML's design divides the language into three main components, with each one having its own set of constructs and concepts: *Process-Structure*, *Activities* and *Definitions*. The *Structure* component of RGML uses a combination of constructs, where each construct expresses a different possible organization of activities within a process, to capture the structure of requirements generation processes. The four main structure constructs RGML uses are:

- **Sequence**: to capture a group of activities the requirements engineer executes in a sequential order
- **Iteration**: to capture a group of activities the requirements engineer executes in a loop
- **Split**: to capture a group of activities from which the requirements engineer chooses only one of the activities to execute
- **Composite**: to capture irregular structures by dividing structures into entry points, connections between activities, and exit points.

The *Activities* component of RGML captures the details of the activities of the process using five main sub-components:

- **Name** : captures the full name of the activity
- **Goal**: captures the goal or goals of the activity
- **Preconditions** : captures the conditions which need to be satisfied before the execution of the activity takes place

- **Steps** : captures the steps the activity recommends in order to reach the set goal of the activity
- **Exceptions**: captures an alternative set of steps (*alternative scenarios*) which the requirements engineer might execute instead of the original set of steps of the activity (*default scenario*), if certain conditions hold.

The *Definitions* component of RGML is responsible for capturing the definitions of four elements, which are referenced throughout the process:

- **Reusable Structures**: the structures that can be reused and referenced within the structure section of RGML
- **Action Types**: the action types the designer references while capturing the details of the activities in the *Activities* component
- **Artifacts:** documents, diagrams, reports or any items the designer references while capturing the details of the activities in the *Activities* component
- **Templates**: documents which can be used as guidelines for the creation of artifacts and which the designer references when he or she is defining artifacts.

In addition to these three main components RGML uses milestones and conditions to enhance its expressive capabilities. Milestones are generic indicators the designer can trigger “On” or “Off” to indicate a number of different events. Conditions in RGML are captured using a flexible and powerful format. Conditions can be based on questions that are displayed to the requirements engineer, or milestones, or artifacts, or any combination of these three elements. Compound conditions in RGML are joined together using the “and” or the “or” Boolean operator. A unique capability of RGML is its use of milestones, conditions, and activity exceptions to capture certain scenarios within the temporal aspect of a process.

6.2 Contributions

The RGML is a flexible language with significant expressive capabilities, which support a formal characterization of the various processes defining requirements generation. The "process-structure" component of RGML permits the description of loops, branching, and sequence constructs. The "activities" component supports the description of individual activities embedded within a requirements generation process. The description of activities includes the specification of alternative activities, conditional-based process flows, the automatic instantiation of software applications, and conditional execution through the use of pre-conditions and milestones .

RGML is unique and can be a benefit to the requirements engineering field in several ways. Because it is descriptive-based, initially RGML provides a foundation for formally thinking about and characterizing the components, activities and flow structure of requirements generation processes. This has an added benefit in that formal characterizations permit a more objective comparison among different generation processes. The long-range plan is to develop an interactive, visually oriented environment, which provides significant guidance and assistance leading to the elicitation, recording, and packaging of a set of quality requirements. RGML is the first step in the realization of that environment because it is a mechanism to specify and describe the requirements generation process in a way that guides and assists the requirements engineer. The development of RGML, therefore, makes an important contribution to the field of requirements engineering.

The RGML is the fruit of our research effort and a new contribution to the software engineering body of knowledge.

6.3 Future Work

An interactive requirements engineering environment is envisioned as the final product of this research and is expected to address a major requirements engineering problem: how to *effectively* guide the requirements engineer through a requirements generation process. With the use of this environment, several problems threatening the success of requirements engineering can be reduced or totally eliminated. As envisioned, this environment can deal with such problems as the requirements engineer's ability to understand the requirement generation process, failure to follow the prescribed requirements engineering process, poor oversight of management for the requirements generation process, and the requirements engineer's ad-hoc composition of sub-processes.

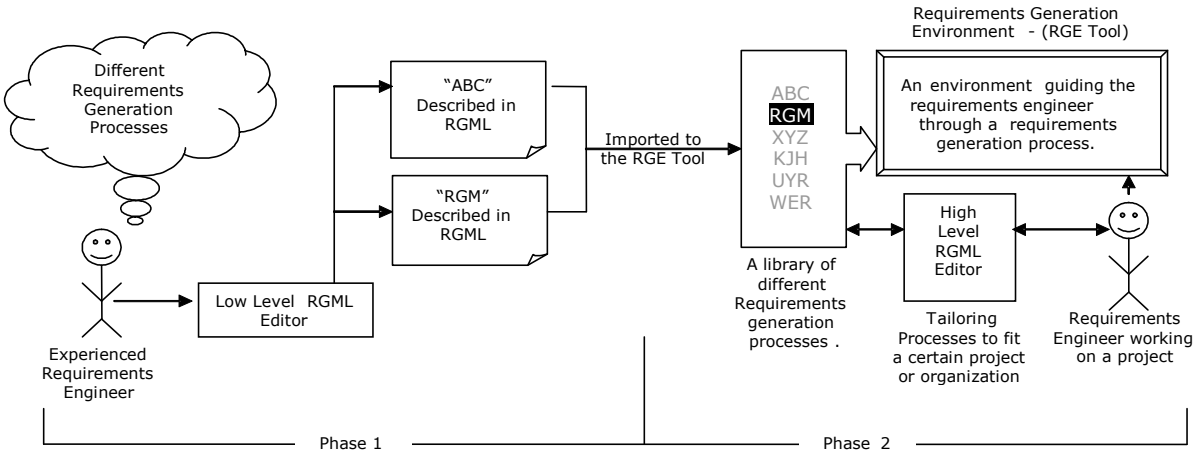

Figure 27. A vision of the Interactive Requirements Engineering Environment

Figure 27 illustrates the vision of a two-phased interactive requirements engineering environment that guides a requirements engineer through the formulation and execution of a structured requirements generation process.

6.3.1 First phase of the environment

Phase one focuses on defining the requirements engineering processes. This phase uses two major components to achieve this definition:

- **Formal Language:** An experienced requirements engineer uses this formal language to formally characterize a requirements generation process. The process can be the one currently being employed in the workplace or an experimental one.

- **Low Level Editor:** The requirements engineer employs a low-level structured editor that assists him or her in the characterization process. Process characterization is achieved through the selection of iconic representations of RGML constructs and concepts, which, in turn, are expanded into RGML code. The authoring tools of the Procedural Markup Language (PML) [Ram 1999] use a similar approach.

After the designer uses the formal language and the low-level editor to express a requirements generation process in RGML, the code is syntactically and semantically validated. When a complete requirements generation process is defined, its definition is stored in a database.

6.3.2 Second phase of the environment

The second phase of the environment addresses executing and refining the requirements engineering processes. This phase also uses two major components to achieve its objectives:

- **Interpreter:** This interpretive tool guides the requirements engineer through the requirements gathering process by reading the selected requirements engineering process from the database, interprets its RGML characterization, and presents a visually based, interactive environment that guides the user (or requirements engineer) through the prescribed process.

- **High-Level Editor:** This editor gives the requirements engineer the flexibility to tailor a process to fit an organization's needs. The editor has a restricted set of capabilities, and through iconic selection and composition permits the manipulation of a visually oriented representation of the requirements generation process. The editor permits tailoring during the requirements generation process or after it is completed. The editor restricts process modifications to those actions which do not violate fundamental principles underlying requirements generation.

As the above section indicates, RGML is only the first step towards the development of a visually oriented environment supporting requirements generation. It is to be followed by the development of:

- a low-level editor that supports the formulation and characterization of requirements generation processes,
- an interpreter that provides an interactive, visually oriented interface, which guides the requirements engineer through the selected requirements generation process, and
- a high-level editor that provides a mechanism for tailoring the requirements generation process.

References

[Ambler 2001]: Ambler, S.W., "Agile Requirements Modeling," *The Official Agile Modeling (AM) Site (2001) ,*http://www.agilemodeling.com

[Arthur 1999]: Arthur, J.D. and Markus K. Groener, 1999, *An Operational Model Supporting the Generation of Requirements that Capture Customer Intent*, Proceedings of the Pacific Northwest Software Quality Conference, Portland OR, October 1999, pp.286-302.

[Berners-Lee 1995]: Berners-Lee, T. and D. Connolly. "*Hypertext Markup Language - 2.0*," RFC 1866, MIT/LCS, November 1995.

[Boehm 1988]: Boehm, B.W., 1988, *A Spiral Model of Software Development and Enhancement*, IEEE Computer, Vol. 21, No. 5, May 1988, pp. 61-72.

[Bray, 1998]: Bray, T., Paoli, J., Sperberg-McQueen, C. M.. *Extensible markup language* (**XML**) 1.0. Recommendation, W3C, February 1998. Available at: http://www.w3.org/TR/1998/REC-xml19980210.

[Brooks 1987]: Brooks F.P., Jr., 1987, *No silver bullet: essence and accidents of software engineering*, Computer, v.20 n.4, p.10-19, April 1987.

[Bryant 1991]: Bryany, B. R., Pan, A., *Formal specification of software systemes using Two-Level Grammar.* COMPASAC '91 fifteenth Annual Intl. Computer Software and Applications Conf., pages 155-160, 1991

[Davis 1993]: Davis, A.M., *Software Requirements: Objects, Functions, & States*, Prentice-Hall, Upper Saddle River, New Jersey, 1993.

[Davis 1999]: Davis A., R. Fairley, E. Yourdon, 1999, *Software Product Planning*, Omni-Vista White Paper #99-002, http://www.rmplace.org/RMWhitePapers.htm, October 1999.

[Dijkstra 1975]: Dijkstra, E. W. "*Guarded Commands, Nondeterminacy, and Formal Derivation of Programs*," Communications of the ACM, Vol. 18, No. 8, pp. 453--457, 1975.

[Du Bois 1995]: Du Bois, P., *The Albert II Language - On the Design and the Use of a Formal Specification Language for Requirements Analysis*, Ph.D. thesis, Dept. of Computer Science, University of Namur, Namur, Belgium, 1995.

[Frincke 1992]: Frincke, Deborah; Wolber, Dave; Fisher, Gene; and Cohen, Gerald: *Requirements Specification Language (RSL) and Supporting Tools*. Nov. 1992.

[Herlea 1999]: Herlea, D.E., Jonker, C.M., Treur, J., and Wijngaards, N.J.E., 1999, *A Formal Knowledge Level Process Model of Requirements Engineering,* In Proceedings of the 12th International Conference on Industrial and Engineering Applications of AI and Expert Systems, IEA/AIE'99.

[IEEE 1993]: IEEE, *Standard **VHDL** Language Reference Manual*. Standard 1076-1993, New York, NY: IEEE, 1993.

[ISO 1986]: International Standards Organization. *Information Processing --- Text and Office Systems --- Standard Generalized Markup Language* (*SGML*), October 1986. ISO 8879.

[Karakostas 2002]: Karakostas, B., Panagiotakis, D., Fakas, G., *Workflow Requirements Modelling Using XML*, Requirements Engineering Journal 7:124-138, Springer-Verlag London 2002

[Lassila 1999]: Lassila, O., Swick, R. R., *Resource description framework* (**RDF**) Model and syntax specification. Recommendation, W3C, February 1999. Available at : http://www.w3.org/TR/1999/RECrdf -syntax-19990222.

[Leffingwell 2000]: Leffingwell D, D. Widrig, 2000, *Managing Software Requirements: A Unified Approac*h, Addison Wesley Publishing Co.

[OMG 1999]: Object Management Group. *OMG Unified Modeling Language Specification UML v1.3.* Technical Report, Document ad/99-06-08, Object Management Group (OMG), 1999

[Pressman 2001]: Pressman R.S., 2001, *Software Engineering: A Practitioner's Approac*h, 5th ed. NewYork, NY: McGraw- Hill, 2001.

[Ram 1999]: Ram A., Catrambone R., Guzdial M., Kehoe C., McCrickard S., Stasko J., *PML: Adding flexability to Multimedia Presentations,* IEEE Multimedia, pp 40-51, April-June 1999

[Robertson 1999]: Robertson, S. and Robertson, J., *Mastering the Requirements Process*. Addison-Wesley, 1999

[Royce 1987]: Royce, W.W., 1970, *Managing the Development of Large Software Systems: Concepts and Techniques*, Proceedings of the Western Electronic Show and Convention (WesCon), Los Angeles, August 1970, pp. 1-9 (Reprinted in the Proceedings of 9th International Conference on Software Engineering, March 1987, Monterey CA, pp. 328 – 338.)

[Saracco 1989]: Saracco, R., Smith, J.R.W., and Reed, R., *Telecommunications Systems Engineering using SDL*, North.Holland Publ, 1989, 632 pgs.

[Schlenoff 1997]: Schlenoff, C., Knutilla, A., and Ray, S., *Proceedings of the Process Specification Language (PSL) Roundtable*, Gaithersburg MD, April 1997. National Institute of Standards and Technology.

[Schlenoff 1998]: Schlenoff, C., Ivester, R., Knutilla, A., *A Robust Ontology for Manufacturing Systems Integration,* Proceedings of the 2nd International Conference on Engineering Design and Automation, Maui, H, August, 1998.

[Schlenoff 2000]: Schlenoff, C., Gruninger M., Tissot, F., Valois, J., Lubell, J., Lee, J., *The Process Specification Language (PSL): Overview and Version 1.0 Specification,* NISTIR 6459, National Institute of Standards and Technology, Gaithersburg, MD (2000).

[Sidky 2002]: Sidky, A.S., Sud, R.R. Bhatia, S, and Arthur, J.D., 2002, *Problem Identification and Decomposition within the Requirements Generation Process*, 6th World Multiconference on Systems, Cybernetics, and Informatics (SCI 2002), Vol. VIII, July 2002, Orlando FL, pp. 333-338.

[Standish 1995]: Standish Group 1995. "Chaos", Standish Research Paper, available at: http://www.standishgroup.com/chaos.html.

[Teichroew 1982]: Teichroew, D., Hersey III, E.A., *"PSL/PSA: a computer-aided technique for structured documentation and analysis of information processing systems"*, in Advanced System Development/ Feasibility Techniques, Wiley, New York, pp 315-329 (1982).

[Thomas 1991]: Thomas, D., Moorby, P., *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.

[WFMC 2002]: Workflow Management Coalition. *Workflow Process Definition Interface - XML Process Definition Language* (**XPDL**), WFMC-TC-1025, Version 1.0 Beta, 2002.

Vita

Ahmed Sidky, the son of Dr. Samy M. Sidky and Hoda Ahmed Farag, was born on August 18th 1980 in Cairo, Egypt.

During his early years, till the age of 8, he lived in Atlanta, Georgia. At Point South Elementary School he was chosen as one of the gifted students.

After moving to live in Cairo, Egypt, he received the Outstanding Academic Presidential Award from President George W. Bush.

He graduated from the October University for Modern Science and Arts (MSA) in August 2000, with a Bachelors' degree in Computer Science. During his Bachelors' degree he received the award for "Ideal student" for the year 2000. Ahmed also received an award for Outstanding Academic Achievement for having the highest GPA in the whole university.

After his Bachelors' degree he was offered a job at the leading Internet Service Provider in Egypt and after 6 months he received an award for "The Most Creative Idea" for the year 2001. During his job as an Internet Solution Developer he was also asked to teach a 2 credit course at his former university.

Subsequently, he has been pursuing a Masters' degree in Computer Science at Virginia Tech, Blacksburg, Virginia. During his Masters' degree, he has served as a Graduate Teaching Assistant and a Graduate Research Assistant, and has been conducting research in the field of Requirements Engineering. This thesis completes his M.S. degree in Computer Science from Virginia Tech.

Ahmed Samy Sidky